

1.1. Lectura / escritura de ficheros



Índice

Introducción a la unidad formativa	3
Objetivos.....	4
Obteniendo información sobre ficheros y carpetas	5
La clase File	5
Obteniendo información de un fichero.....	8
Obteniendo información de una carpeta	9
Los flujos de datos.....	13
Clases que representan flujos de datos.....	13
Flujos de datos en formato Unicode de 16 bits.....	14
Flujos de bytes (información binaria)	15
Lectura y escritura de ficheros.....	17
Escritura en un fichero de texto	17
Lectura de un fichero de texto.....	22
Escritura en un fichero binario	25
Lectura de un fichero binario	29
Lectura de teclado con InputStreamReader y BufferedReader	32
La clase System.....	32
Gestión de excepciones en la lectura / escritura	33
La clase scanner	35
Introducción a la clase Scanner	35
Lectura de cadenas de texto	35
Leer un <i>String</i> palabra a palabra	36
Leer un <i>String</i> línea a línea	36
Leer un <i>String</i> según separador personalizado.....	37
Lectura de un <i>String</i> con datos numéricos	37
Lectura desde teclado.....	38
Lectura de ficheros de texto.....	40
Despedida	41
Resumen.....	41

El contenido de esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.

Introducción a la unidad formativa

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

1. Obtener información sobre archivos y carpetas haciendo uso de la clase Java *File*.
2. Escribir programas Java que realicen operaciones de lectura y escritura de ficheros mediante el uso de la jerarquía de clases manejadoras de flujos de datos.
3. Gestionar las excepciones que pueden producirse durante las operaciones de lectura y escritura de ficheros.
4. Escribir programas Java utilizando la clase *Scanner* para leer cadenas de texto, introducir datos por teclado o leer archivos de texto.

¡Ánimo y adelante!

Obteniendo información sobre ficheros y carpetas

La clase File

La clase *File*, situada en el paquete *java.io*, nos permite obtener información sobre archivos y carpetas.

Cada objeto *File* construido vendrá a representar a un determinado archivo o a una determinada carpeta dentro del sistema de archivos.

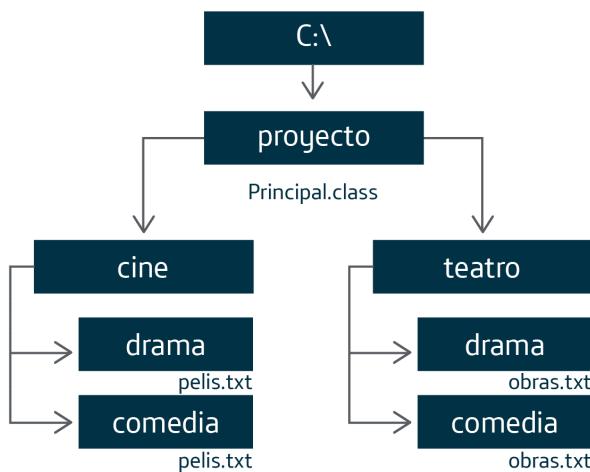
El constructor de la clase **File** está sobrecargado, de manera que podemos crear un nuevo objeto *File* con cualquiera de estos tres formatos:

- *File (String path)*.
- *File (String path, String nameFile or path)*.
- *File (File path, String nameFile or path)*.

Donde *Path* representa la ruta dentro de la estructura de carpetas y podrá especificarse de forma absoluta o relativa.

- Una ruta absoluta se especifica a partir de la raíz del disco.
- Una ruta relativa se especifica a partir de la ubicación actual, es decir, a partir de la ubicación de la clase Java que intenta acceder a dicha ruta.

Imagina que vas a ejecutar un programa Java situado en **C:\ proyecto** con el nombre **Principal.class**.



Dentro de la carpeta *proyecto* además existen las carpetas *cine* y *teatro* con las subcarpetas *drama* y *comedia*, tal y como puedes apreciar en la imagen.

Dentro de cada carpeta *drama* y *comedia* además tenemos ficheros de texto.

Utilizaremos esta estructura como ejemplo para mostrar varias formas de crear objetos *File*.

A continuación veremos varios ejemplos de construcción de objetos de la clase *File* utilizando tanto rutas absolutas como relativas.

Usando rutas absolutas

```
File fich = new File("C:/proyecto/cine/drama/pelis.txt");
```

El objeto *fich* representa el fichero especificado en el argumento. Hemos utilizado el primer constructor.

```
File fich = new File("C:/proyecto/cine/drama", "pelis.txt");
```

Este ejemplo es igual que el anterior, pero estamos utilizando el segundo constructor: *File (String path, String nameFile)*.

```
File carp = new File("C:/proyecto/cine/drama");
```

El objeto *carp* representa a la carpeta *drama*. Hemos utilizado el primer constructor.

```
File carp = new File("C:/proyecto/", "cine/drama");
```

Este ejemplo es igual que el anterior, pero estamos utilizando el segundo constructor: *File (String path, String subPath)*.

```
File carp = new File("C:/proyecto/cine/drama");
File fich = new File(carp, "pelis.txt");
```

En este ejemplo estamos creando el objeto *fich*, que representa un fichero a partir del objeto *carp*, que representa la carpeta donde está ubicado el fichero. Estamos utilizando el tercer constructor para crear el objeto *fich*: *File (File path, String nameFile)*.

CURIOSIDAD: usamos el símbolo / en lugar del símbolo \ dentro de la ruta porque el símbolo \ es un carácter de escape especial para Java y nos ocasionaría error de compilación. Recuerda que en varias ocasiones has incluido la combinación "\n" en una cadena de texto para generar un retorno de carro, esto es porque el símbolo \ va seguido de alguno de los caracteres especiales que tienen un significado específico (un retorno de carro en el caso de la n).

Podemos especificar la ruta de dos formas distintas:

C:/proyecto/cine/drama o C:\\proyecto\\cine\\drama

En el segundo caso hemos colocado el símbolo \ como un carácter especial de escape y así no tenemos errores de compilación.

[Obtén información sobre los caracteres especiales de escape Java.](#)

Usando rutas relativas

Ahora vamos a presentar exactamente los mismos ejemplos anteriores pero con rutas relativas, es decir, a partir de la ubicación del programa, que en el ejemplo de la imagen es la carpeta *C:\Proyecto*. En este caso usaremos el carácter de escape \ para que te acostumbres a los dos sistemas, aunque podrías hacerlo con el carácter / igualmente.

```
File fich = new File("cine\\drama\\pelis.txt");
```

El objeto *fich* representa el fichero especificado en el argumento.

```
File fich = new File("cine\\drama", "pelis.txt");
```

Este ejemplo es igual que el anterior, pero estamos utilizando el segundo constructor: *File (String path, String nameFile)*.

```
File carp = new File("cine\\drama");
```

El objeto *carp* representa a la carpeta *drama*. Hemos utilizado el primer constructor.

```
File carp = new File("cine\\drama");
File fich = new File(carp, "pelis.txt");
```

En este ejemplo estamos creando el objeto *fich*, que representa un fichero a partir del objeto *carp*, que representa la carpeta donde está ubicado el fichero. Estamos utilizando el tercer constructor para crear el objeto *fich*: *File (File path, String nameFile)*.

¿Y qué podemos hacer con un objeto *File*?

La clase *File* nos permite las siguientes operaciones:

1. **Obtener información sobre archivos:** número de bytes que ocupa, propiedades del archivo (si es de solo lectura, oculto, etc.), ruta donde se encuentra, etc.
2. **Obtener información sobre carpetas:** propiedades de la carpeta, archivos y subcarpetas que contiene, etc.
3. **Borrar archivos y carpetas.**
4. **Crear archivos y carpetas.**

Aunque la clase *File* permita borrar y crear archivos y carpetas, no permite operaciones de lectura y escritura.

¿Qué es lo que NO permite la clase *File*?

1. **No permite leer** el contenido de un fichero. Para eso están los flujos de datos de lectura que estudiaremos en otro apartado de esta misma unidad.
2. **No permite escribir** dentro de un fichero. Para eso están los flujos de datos de escritura que estudiaremos en otro apartado de esta misma unidad.

Obteniendo información de un fichero

Ya tengo mi objeto *File*.

¿Ahora qué hago con él?

Prueba a ejecutar este pequeño programa. Crea un objeto *File* que representa a un archivo llamado *pelis.txt* y después obtiene la siguiente información: número de bytes que ocupa, nombre del archivo, ruta, propiedades del fichero (si es oculto, si se puede escribir en él, si se puede leer).

```
import java.io.File;

public class Principal {
    public static void main(String args[]) {
        File fich = new File("C:/proyecto/cine/drama/pelis.txt");
        if (fich.exists()) {
            System.out.println("Existe el fichero");
            System.out.println("Nº de bytes que ocupa: " + fich.length());
            System.out.println("Nombre de archivo: " + fich.getName());
            System.out.println("Ruta: " + fich.getPath());
            System.out.println("¿Es un fichero oculto? " + fich.isHidden());
            System.out.println("¿Está permitida la escritura? " + fich.canWrite());
            System.out.println("¿Está permitida la lectura? " + fich.canRead());
        } else {
            System.out.println("El fichero no existe");
        }
    }
}
```

La clase *File* también nos permite crear o eliminar un fichero.

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File fich = new File("C:/proyecto/cine/drama/pelisdeterror.txt");
        boolean ok = fich.createNewFile();
        if (ok)
            System.out.println("El fichero se ha creado con éxito");
        else
            System.out.println("El fichero no ha podido crearse");
    }
}
```

Como has podido comprobar por el ejemplo anterior, podemos instanciar un objeto *File* que represente un fichero que no existe y después crearlo con el método *createNewFile()*, que creará físicamente el fichero con el nombre y ruta especificada en el constructor de *File*. El método *createNewFile()* devuelve *true* si el fichero se ha creado y *false* de lo contrario.

Si pruebas a ejecutar el programa dos veces la primera mostrará el mensaje "El fichero se ha creado con éxito", siempre y cuando la ruta especificada exista. La segunda vez mostrará el mensaje "El fichero no ha podido crearse", ya que no se puede crear un fichero que ya existe.

Prueba a ejecutar una vez más el programa, pero ahora con una ruta que no exista.

Comprueba que se desencadena una excepción en tiempo de ejecución.

El método *createNewFile()* puede provocar excepciones de tipo ***IOException***; debemos encerrarlo en un *try ... catch* o bien propagar la excepción con un *throws*.

Ten en cuenta que la operación de crear un fichero en disco puede tener varias situaciones de excepción:

- La ruta especificada no existe.
- Intentamos crear un fichero en una carpeta que está protegida contra escritura.
- El disco utilizado está deteriorado o lleno.

Ahora vamos a eliminar el fichero que acabamos de crear:

```
import java.io.File;

public class Principal {
    public static void main(String args[]) {
        File fich = new File("C:/proyecto/cine/drama/pelisderror.txt");
        boolean ok = fich.delete();
        if (ok)
            System.out.println("El fichero se ha borrado con éxito");
        else
            System.out.println("El fichero no ha podido borrarse");
    }
}
```

El método *delete()* elimina el fichero y devuelve *true* si la operación se ha completado con éxito, de lo contrario devuelve *false*.

Obteniendo información de una carpeta

En esta ocasión vamos a trabajar con un objeto *File* que representa una carpeta.

Pega este código en un proyecto Eclipse y ejecútalo:

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\\\proyecto\\\\cine\\\\drama");
        if (carp.exists()) {
            System.out.println("Existe la carpeta");
            System.out.println("¿Tiene permisos de escritura? " + carp.canWrite());
            String[] contenido = carp.list();
            System.out.println("Archivos o carpetas que contiene: " +
            contenido.length);
            for (String nombre : contenido) {
                System.out.println(nombre);
            }
        } else
            System.out.println("No existe la carpeta");
    }
}
```

Vamos a analizar el programa poco a poco:

```
System.out.println("¿Tiene permisos de escritura? " + carp.canWrite());
```

El método ***canWrite()*** devuelve *true* si la carpeta tiene permisos de escritura, es decir, no está protegida como "solo lectura".

```
String[] contenido = carp.list();
System.out.println("Archivos o carpetas que contiene: " + contenido.length);
```

El método *list()* devuelve un *array* de objetos *String* con los nombres de los archivos o subcarpetas contenidos en la carpeta que representa el objeto *carp*. El *array* devuelto lo estamos guardando en la variable *contenido*, que será un *array*. La propiedad *length* del *array* contiene el número de elementos, que en este caso coincide con el número de archivos o carpetas.

```
for (String nombre : contenido) {
    System.out.println(nombre);
}
```

Por último, estamos utilizando una estructura *for each* para recorrer los elementos del *array* y así mostrar en pantalla los nombres de los archivos y carpetas.

Vamos a dar otro paso más.

Si podemos obtener un *array* con los nombres de archivos y carpetas, también podremos utilizar estos nombres para construir nuevos objetos *File* para obtener más información sobre cada uno de ellos. ¡Vamos a comprobarlo! Accederemos a una carpeta que tenga más contenido, por ejemplo la carpeta *Windows*, que casi seguro está situada en *C:\Windows*.

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\windows");
        if (carp.exists()) {
            System.out.println("Existe la carpeta");
            System.out.println("¿Tiene permisos de escritura? " + carp.canWrite());
            String[] contenido = carp.list();
            System.out.println("Archivos o carpetas que contiene: " + contenido.length);
            for (String nombre : contenido) {
                File f = new File(carp.getPath(), nombre);
                if (f.isDirectory()) {
                    System.out.println(nombre + ", " + " carpeta");
                }
                else {
                    System.out.println(nombre + ", " + " fichero, " + f.length() + " bytes");
                }
            }
        }
        else
            System.out.println("No existe la carpeta");
    }
}
```

Este ejemplo es muy parecido al anterior, pero ahora estamos accediendo a la carpeta *Windows*, comprobando por cada elemento si se trata de una carpeta o archivo. En caso de ser un archivo muestra el número de bytes que ocupa.

La clase *File* también sirve para crear y eliminar carpetas.

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\prueba");
        boolean ok = carp.mkdir();
```

```
        if (ok)
            System.out.println("La carpeta se ha creado con éxito");
        else
            System.out.println("La carpeta no ha podido crearse");
    }
}
```

El método `mkdir()` crea la carpeta representada por el objeto `File` si no existe. Devuelve `true` si la carpeta se ha podido crear con éxito y `false` de lo contrario.

Ahora vamos a borrar la carpeta que acabamos de crear:

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

public class Principal {
    public static void main(String args[]) throws IOException {
        File carp = new File("C:\\prueba");
        boolean ok = carp.delete();
        if (ok)
            System.out.println("La carpeta se ha borrado con éxito");
        else
            System.out.println("La carpeta no ha podido borrarse");
    }
}
```

El método `delete()` elimina la carpeta. Devuelve `true` si la carpeta se ha eliminado con éxito y `false` de lo contrario. El método `delete()` no permite eliminar una carpeta que tenga algo dentro, es decir, no puede contener ningún archivo ni subcarpeta.

Los flujos de datos

Clases que representan flujos de datos

Toda la información que se transmite a través de un ordenador fluye desde una entrada hacia una salida. Para transmitir información, Java utiliza unos objetos especiales denominados *streams* (flujos o corrientes).

Toda operación de lectura o escritura de ficheros requiere del uso de un flujo de datos o *stream*.



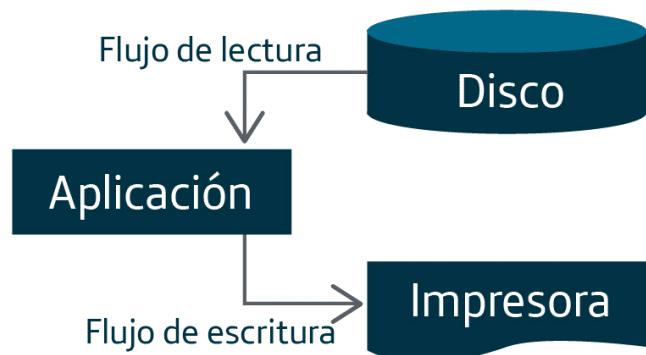
Los *stream* permiten transmitir secuencias ordenadas de datos desde un origen a un destino. El origen y el destino puede ser un fichero, un *String* o un dispositivo (lectura de teclado, escritura en pantalla).

Java dispone de dos grupos de flujos de datos:

1. **Flujos de entrada o lectura (*input streams*):** los datos fluyen desde el fichero o dispositivo hacia el programa.
2. **Flujos de salida o escritura (*output streams*):** los datos fluyen desde el programa hacia el fichero o dispositivo.

Java no dispone de flujos de entrada / salida

Java no dispone de clases que representen flujos de lectura y escritura. Si necesitamos leer y escribir de un fichero necesitamos dos flujos distintos: un flujo de entrada o lectura y otro de salida o escritura.



Organigrama de lectura y escritura de archivos.

Todo proceso de lectura o escritura de datos consta de tres pasos:

1. **Abrir el flujo** de datos de lectura o de escritura.
2. **Leer o escribir** datos a través del flujo abierto.
3. **Cerrar** el flujo de datos.

Las clases manejadoras de flujos de datos

Todas las clases que representan flujos de datos están ubicadas en el paquete `java.io`. Dentro del paquete `java.io` disponemos de varias clases para representar flujos de datos. Están organizadas en dos grandes grupos:

1. **Flujos de datos en formato Unicode de 16 bits:** derivados de las clases abstractas `Reader` y `Writer`.
2. **Flujos de bytes (información binaria):** derivados de las clases abstractas `InputStream` y `OutputStream`.

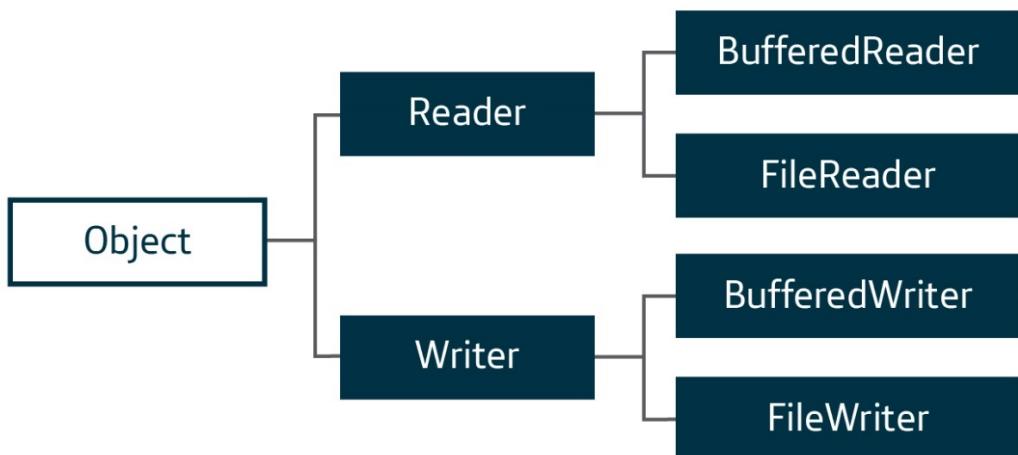
En los dos siguientes apartados estudiaremos más detenidamente las clases más importantes del paquete `java.io`.

Flujos de datos en formato Unicode de 16 bits

Todas las clases que representan flujos de datos (*streams*) en formato Unicode de 16 bits derivan de las clases abstractas `Reader` y `Writer`. En la imagen hemos reflejado las clases más importantes dentro de esta categoría.

Los flujos de datos, además de diferenciarse según sean de entrada o salida, también se distinguen por su cercanía al dispositivo. En este sentido hay dos tipos de flujos de datos:

- **Iniciadores:** vuelcan o recogen datos directamente del dispositivo.
- **Filtros:** se sitúan entre el *stream* iniciador y el programa.



- **Reader**

Clase abstracta de la que derivan todas las clases que representan flujos de entrada de caracteres Unicode de 16 bits.

- **Writer**

Clase abstracta de la que derivan todas las clases que representan flujos de salida de caracteres Unicode de 16 bits.

- **BufferedReader**

Permite mejorar la velocidad de transmisión en la lectura de un fichero proporcionando un *buffer*. Entra dentro de la categoría de filtro y trabaja en colaboración con un objeto *FileReader*.

- **FileReader**

Permite leer caracteres de un fichero. Es iniciador y puede trabajar en conjunto con la clase *BufferedReader*, que actúa como filtro y mejora la eficiencia de la lectura.

- **BufferedWriter**

Permite mejorar la velocidad de escritura en un fichero proporcionando un *buffer*. Entra dentro de la categoría de filtro y trabaja en colaboración con la clase *FileWriter*.

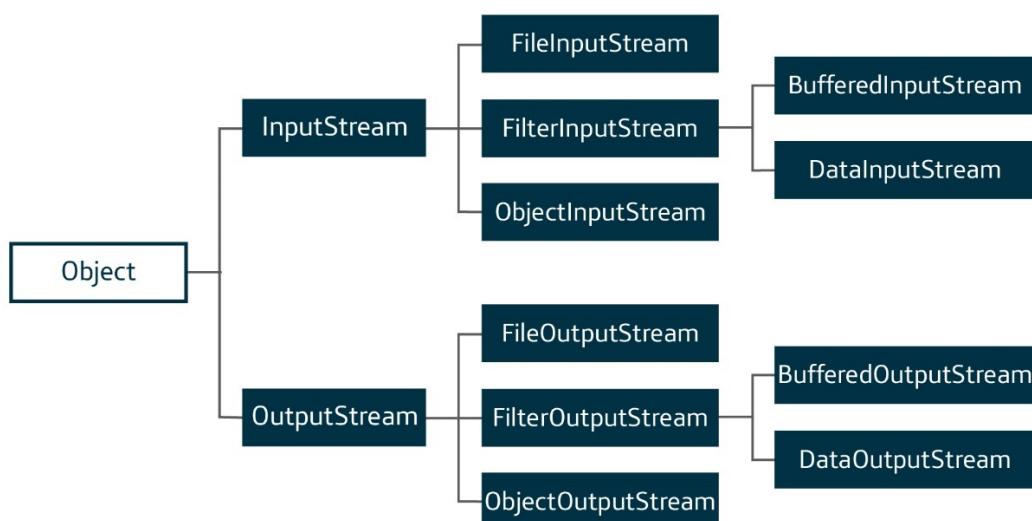
- **FileWriter**

Permite escribir caracteres en un fichero. Es iniciador y puede trabajar en conjunto con la clase *BufferedWriter*, que actúa como filtro y mejora la eficiencia de la escritura.

Flujos de bytes (información binaria)

Todas las clases que representan flujos de datos (*streams*) en formato binario derivan de las clases abstractas ***InputStream*** y ***OutputStream***. En la imagen hemos reflejado las clases más importantes dentro de esta categoría. Igual que ocurría con los flujos de caracteres Unicode, los flujos de bytes también se subdividen en:

- **Iniciadores:** vuelcan o recogen datos directamente del dispositivo.
- **Filtros:** se sitúan entre el *stream* iniciador y el programa.



- **InputStream**

Clase abstracta de la que derivan todas las clases que representan flujos de entrada de bytes.

- **FileInputStream**

Permite leer bytes de un fichero. Actúa como iniciador.

- **FilterInputStream**

Clase base de la que derivan las siguientes subclases que actúan como filtro, mejorando las operaciones de lectura: *BufferedInputStream* y *DataInputStream*.

- **ObjectInputStream**

Permite la lectura de objetos guardados en disco. Actúa como filtro y requiere un iniciador, por ejemplo un *FileInputStream*.

- **BufferedInputStream**

Permite mejorar la eficiencia de la lectura de un fichero proporcionando un *buffer*. Trabaja en colaboración con un objeto iniciador, por ejemplo un *FileInputStream*.

- **DataInputStream**

DataInputStream permite leer datos de un fichero recogiéndolos directamente como tipos de datos primitivos (*int*, *float*, *double*, etc.). Actúa como filtro y trabaja en colaboración con otra clase iniciadora como *FileInputStream*.

- **OutputStream**

Clase abstracta de la que derivan todas las clases que representan flujos de salida de bytes.

- **FileOutputStream**

Permite escribir bytes en un fichero. Actúa como iniciador.

- **FilterOutputStream**

Clase base de la que derivan las siguientes subclases que actúan como filtro, mejorando las operaciones de escritura: *BufferedOutputStream* y *DataOutputStream*.

- **ObjectOutputStream**

Permite la escritura de objetos en disco. Actúa como filtro y requiere un iniciador, por ejemplo un *FileOutputStream*.

- **BufferedOutputStream**

Permite mejorar la eficiencia de la escritura en un fichero proporcionando un *buffer*. Trabaja en colaboración con un objeto iniciador, por ejemplo un *FileOutputStream*.

- **DataOutputStream**

Permite escribir datos en un fichero directamente como tipos de datos primitivos (*int*, *float*, *double*, etc.). Actúa como filtro y trabaja en colaboración con otra clase iniciadora como *FileOutputStream*.

Lectura y escritura de ficheros

Escritura en un fichero de texto

En este apartado tendrás oportunidad de practicar con los flujos de datos para la escritura en ficheros de texto, principalmente en formato Unicode de 16 bits.

Generalmente las operaciones de lectura / escritura requieren de un objeto iniciador que se comunique directamente con el dispositivo y un filtro con el que realizar la lectura / escritura eficientemente.

Escritura en formato Unicode con *FileWriter* y *BufferedWriter*

En el siguiente programa utilizaremos las clases *FileWriter* (iniciador) y *BufferedWriter* (filtro) para escribir el título de tres películas en un fichero de texto. Para ello realizaremos los tres pasos típicos asociados a cualquier tarea de escritura en ficheros:

1. Abrir fichero para escritura.
2. Escribir líneas en el fichero.
3. Cerrar el fichero.

```
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileWriter file;
        try {
            file = new FileWriter("C:\\cine\\peliculas.txt");
        } catch (IOException e) {
            System.out.println("No se puede abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Abrir buffer y escribir tres líneas
        BufferedWriter buffer = new BufferedWriter(file);
        try {
            buffer.write(";Bienvenido, Mister Marshall!");
            buffer.newLine();
            buffer.write("Con la muerte en los talones");
            buffer.newLine();
            buffer.write("Muerte de un ciclista");
        }
    }
}
```

```
        buffer.newLine();
    } catch (IOException e) {
        System.out.println("Error al escribir en el fichero");
        System.out.println(e.getMessage());
    }

    // Cerrar el buffer y el fichero
    try {
        buffer.close();
        file.close();
    } catch (IOException e) {
        System.out.println("Error al cerrar el fichero");
        System.out.println(e.getMessage());
    }
}
```

Ahora vamos a analizar las partes más importantes del programa:

- **file = new FileWriter("C:\\\\cine\\\\peliculas.txt");**

Al construir un objeto de la clase *FileWriter* estamos abriendo el fichero *peliculas.txt*, dejándolo preparado para escritura.

- **BufferedWriter buffer = new BufferedWriter(file);**

Las operaciones de escritura las realizaremos a través del objeto *buffer* de la clase *BufferedWriter*, que nos proporciona métodos para la escritura eficiente. El constructor de la clase *BufferedWriter* requiere un objeto *FileWriter*.

- **buffer.write(";Bienvenido, Mister Marshall!");**

El método *write(String texto)* de la clase *BufferedWriter* escribe texto en el fichero.

- **buffer.newLine();**

El método *newLine()* de la clase *BufferedWriter* escribe un retorno de carro en el fichero.

- **buffer.close();**

file.close();

Por último tenemos que cerrar los objetos *BufferedWriter* y *FileWriter*.

Habráis observado que, aunque el fichero ***peliculas.txt*** no existía previamente, el constructor de la clase ***FileWriter*** lo ha creado.

La sentencia:

file = new FileWriter("C:\\cine\\peliculas.txt");

no solo nos ha permitido abrir el fichero para escritura, sino que también lo ha creado físicamente. Si pruebas a ejecutar varias veces el programa verás que no se van añadiendo nuevas líneas, siempre hay tres. Cada vez que ejecutamos vuelve a crear el fichero sobrescribiendo el anterior. **Si lo que deseamos es añadir líneas a un fichero existente solo tenemos que pasar un argumento más al constructor de *FileWriter* con el valor *true*.**

```
file = new FileWriter("C:\\cine\\peliculas.txt", true);
```

Así abrimos el fichero para añadir nuevas líneas. Si el fichero no existe, lo crea, pero si existe lo abre para añadir.

El método *write* de la clase *BufferedWriter* también puede recibir un número entero, que escribirá en el fichero el carácter asociado en Unicode con dicho número.

```
buffer.write(65);
```

Esta sentencia escribiría en el fichero una 'A', carácter asociado al valor numérico 65. Pon a prueba este ejemplo, que escribe en un fichero los caracteres Unicode asociados a los números 0 a 255.

```
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {
        // Abrir fichero para escritura
        FileWriter file;
        try {
            file = new FileWriter("C:\\cine\\caracteres.txt");
        } catch (IOException e) {
            System.out.println("No se puede abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Abrir buffer y escribir tres líneas
        BufferedWriter buffer = new BufferedWriter(file);
        try {
            for (int i=0; i<=255; i++) {
                buffer.write(i+": "); // Escritura de un String. La operación i+": " genera un String
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

```

                buffer.write(i); // Escritura del carácter asociado al
valor de i.
                buffer.newLine();
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el buffer y el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }

    }

}
}

```

También es posible escribir en ficheros de texto con un flujo de datos en formato binario, pero en ese caso hay que escribir uno a uno cada byte que corresponda al texto.

Pon a prueba el siguiente ejemplo:

```

import java.io.FileOutputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileOutputStream file;
        try {
            file = new FileOutputStream("C:\\cine\\peliculas2.txt");
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }
        String texto = "Con la muerte en los talones";

        // Escribir el texto en el fichero carácter a carácter.
        try {
            for (int i=0; i<texto.length(); i++) {
                file.write(texto.charAt(i));
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            file.close();
        }
}

```

```
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Observa que hemos realizado todo el proceso de escritura exclusivamente con un objeto iniciador (*FileOutputStream*). También podemos hacer que este objeto iniciador actúe en colaboración con un filtro para mejorar el rendimiento proporcionando un *buffer*. El siguiente programa es igual que el anterior pero incorpora un *buffer*.

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

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileOutputStream file;
        BufferedOutputStream buffer;
        try {
            file = new FileOutputStream("C:\\cine\\peliculas2.txt");
            buffer = new BufferedOutputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }
        String texto = "Con la muerte en los talones";

        // Escribir el texto en el fichero carácter a carácter.
        try {
            for (int i=0; i<texto.length(); i++) {
                buffer.write(texto.charAt(i));
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Lectura de un fichero de texto

En este apartado tendrás oportunidad de practicar con los flujos de datos para la lectura de ficheros de texto, principalmente en formato Unicode de 16 bits.

Comenzaremos por utilizar las clases **FileReader** (iniciador) y **BufferedReader** (filtro) para realizar la lectura del fichero *peliculas.txt* creado en el apartado anterior.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para lectura
        FileReader file;
        try {
            file = new FileReader("C:\\\\cine\\\\peliculas.txt");
        } catch (IOException e) {
            System.out.println("No se puede abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Abrir buffer y escribir tres líneas
        BufferedReader buffer = new BufferedReader(file);
        String linea="";
        try {
            linea = buffer.readLine();
            while (linea!=null) {
                System.out.println(linea);
                linea = buffer.readLine();
            }
        } catch (IOException e) {
            System.out.println("Error al leer el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el buffer y el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Analicemos ahora las partes más importantes del programa:

- **file = new FileReader("C:\\cine\\peliculas.txt");**

Con esta sentencia estamos abriendo el fichero *peliculas.txt* para lectura, desencadenando una excepción del tipo *FileNotFoundException* si el fichero no existe o cualquier otra excepción de tipo *IOException* si se produce otro tipo de error que no permita la apertura del fichero.

- **linea = buffer.readLine();**

Con esta sentencia estamos leyendo la siguiente línea del fichero de manera secuencial y guardándola en la variable *linea*. Cuando ya no hay más líneas para leer devuelve *null*; esa es la razón por la que necesitamos una estructura *while* con la condición *linea!=null*.

- **buffer.close();**

file.close();

Finalmente debemos desbloquear el fichero cerrando los flujos de datos de filtro e iniciador.

También podemos leer el contenido de un fichero de texto con un objeto de la clase *FileInputStream*, pero como se trata de un flujo de datos en formato binario, hay que leer uno a uno los bytes correspondientes a cada uno de los caracteres.

Ponlo en práctica ejecutando este programa:

```
import java.io.FileInputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileInputStream file;
        try {
            file = new FileInputStream("C:\\cine\\peliculas.txt");
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Leer contenido del fichero carácter a carácter
        int caracter;
        try {
            caracter = file.read(); // Lee un byte y devuelve -1 si es
final de fichero
            while (caracter!=-1) {
                System.out.print((char)caracter);
                caracter = file.read();
            }
        } catch (IOException e) {
```

```

        System.out.println("Error al leer el fichero");
        System.out.println(e.getMessage());
    }

    // Cerrar el fichero
    try {
        file.close();
    } catch (IOException e) {
        System.out.println("Error al cerrar el fichero");
        System.out.println(e.getMessage());
    }

}

}

```

Analicemos las partes más importantes del programa:

- **file = new FileInputStream("C:\\cine\\peliculas.txt");**
Con esta línea abrimos el fichero para lectura.
- **caracter = file.read();**
Esta sentencia lee el próximo carácter en orden secuencial y devuelve un número entero con el código Unicode asociado al carácter. Cuando llega al final de fichero devuelve -1, esa es la razón por la que usamos una estructura *while* cuya condición es que la variable *caracter* sea distinta de -1. A la hora de mostrar el carácter en pantalla hacemos conversión a tipo *char* para que se vayan mostrando los caracteres y no los códigos.
- **file.close();**
Con esta sentencia cerramos el fichero.

Observa que en esta ocasión hemos realizado la lectura utilizando exclusivamente un objeto de tipo iniciador (*FileInputStream*). También es posible utilizar un objeto que actúe de filtro proporcionando un método más eficiente de lectura. El siguiente programa funciona exactamente igual que el anterior, pero incorpora un filtro.

```

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) {

        // Abrir fichero para escritura
        FileInputStream file;
        BufferedReader buffer;
        try {
            file = new FileInputStream("C:\\cine\\peliculas.txt");
            buffer = new BufferedReader(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
        }
    }
}

```

```
        return;
    }

    // Leer contenido del fichero carácter a carácter
    int caracter;
    try {
        caracter = buffer.read();
        // Lee un byte y devuelve -1 si es final de fichero
        while (caracter!= -1) {
            System.out.print((char)caracter);
            caracter = buffer.read();
        }
    } catch (IOException e) {
        System.out.println("Error al leer el fichero");
        System.out.println(e.getMessage());
    }

    // Cerrar el fichero
    try {
        buffer.close();
        file.close();
    } catch (IOException e) {
        System.out.println("Error al cerrar el fichero");
        System.out.println(e.getMessage());
    }
}
```

Escritura en un fichero binario

Los ficheros binarios están formados por secuencias de bytes, pueden contener datos de tipo elemental (*int*, *float*, *double*, etc.) u objetos.

El formato binario de datos es muy útil cuando se quieren representar datos en forma de registros y campos, como si de una base de datos se tratara. En esta ocasión, como ejemplo, vamos a crear un programa que permita añadir los datos de los artículos disponibles en un almacén. Comenzaremos por revisar los pasos que tenemos que seguir para lograrlo:

- Crear una clase llamada *Producto*, que servirá para representar la estructura de cada uno de los artículos del almacén y utilizarla para crear todos los objetos *Producto* que sean necesarios.
- Crearemos un objeto de la clase *FileOutputStream*, que servirá como flujo iniciador para abrir el fichero *almacen.dat*.
- Crearemos un objeto de la clase *DataOutputStream*, que servirá como filtro proporcionando un *buffer* de escritura y lo vincularemos al objeto *FileOutputStream*.
- Escribiremos registros en el *buffer* proporcionado por el objeto *DataOutputStream*.

- Cerraremos los flujos *DataOutputStream* y *FileOutputStream*.

Crea un proyecto Eclipse con el nombre que deseas y crea la clases *Producto* y *Principal*, cuyo código puedes copiar y pegar desde aquí:

```
public class Producto {  
    private String nombre;  
    private float precio;  
    private float unidadesEnExistencia;  
  
    public Producto(String nombre, float precio, float unidadesEnExistencia) {  
        this.nombre = nombre;  
        this.precio = precio;  
        this.unidadesEnExistencia = unidadesEnExistencia;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
    public float getPrecio() {  
        return precio;  
    }  
    public float getUnidadesEnExistencia() {  
        return unidadesEnExistencia;  
    }  
  
    @Override  
    public String toString() {  
        return nombre + " Stock: " + this.unidadesEnExistencia + " Precio: "  
+ this.precio;  
    }  
}
```

Clase que sirve para representar la estructura de cada artículo del almacén.

```
import java.io.DataOutputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
public class Principal {  
  
    public static void main(String[] args) {  
  
        // Creación de 3 objetos producto  
        Producto p1 = new Producto("Manzanas Royal Gala", 2.50f, 7f);  
        Producto p2 = new Producto("Dátiles de la tía Julita", 3.25f, 12f);  
        Producto p3 = new Producto("Mandarinas Clementinas", 2.20f, 25f);  
  
        FileOutputStream fichero;  
        DataOutputStream escritor;
```

```
// Apertura del fichero almacen.dat
try {
    fichero = new FileOutputStream("almacen.dat", true);
    escritor = new DataOutputStream (fichero);
} catch (IOException e) {
    System.out.println("No se ha podido abrir el fichero
almacen.dat");
    System.out.println(e.getMessage());
    return;
}

// Escribir datos en el fichero almacen.dat
try {
    escritor.writeUTF(p1.getNombre());
    escritor.writeFloat(p1.getPrecio());
    escritor.writeFloat(p1.getUnidadesEnExistencia());

    escritor.writeUTF(p2.getNombre());
    escritor.writeFloat(p2.getPrecio());
    escritor.writeFloat(p2.getUnidadesEnExistencia());

    escritor.writeUTF(p3.getNombre());
    escritor.writeFloat(p3.getPrecio());
    escritor.writeFloat(p3.getUnidadesEnExistencia());

} catch (IOException e) {
    System.out.println("Ha ocurrido un error al escribir datos en
el fichero");
    System.out.println(e.getMessage());
}

try {
    escritor.close();
    fichero.close();
} catch (IOException e) {
    System.out.println("Ha ocurrido un error al cerrar el
fichero");
    System.out.println(e.getMessage());
}
}
```

Guardar tres artículos en el fichero *almacen.dat*.

Vamos a repasar las acciones que hemos llevado a cabo:

Hemos creado tres objetos de la clase *Producto* con los datos que posteriormente guardaremos en el fichero *almacen.dat*

```
// Creación de 3 objetos producto
Producto p1 = new Producto("Manzanas Royal Gala",2.50f,7f);
Producto p2 = new Producto("Dátiles de la tía Julita",3.25f,12f);
Producto p3 = new Producto("Mandarinas Clementinas",2.20f,25f);
```

Hemos creado un objeto de la clase *FileOutputStream*

```
fichero = new FileOutputStream("almacen.dat", true);
```

La construcción del objeto *FileOutputStream*, que actúa como flujo de datos iniciador, nos ha dejado el archivo *almacen.dat* abierto para escritura. Con el segundo argumento asignado a *true* logramos que, si el fichero ya existe, agregue los nuevos datos sin sobrescribir lo anterior.

Como no hemos especificado ninguna ruta para el fichero *almacen.dat*, se creará en la misma carpeta donde se encuentra el proyecto Eclipse.

Hemos creado un objeto *DataOutputStream*.

```
escritor = new DataOutputStream (fichero);
```

Construimos el objeto *DataOutputStream* (filtro) pasando como argumento el nuevo objeto *FileOutputStream* para proporcionar un sistema eficiente que permite guardar en disco datos de tipo elemental. En nuestro caso guardaremos un texto en formato UTF y dos valores tipo *float*.

Escribimos registros con ayuda del objeto *DataOutputStream* (*buffer*)

```
escritor.writeUTF(p1.getNombre());
escritor.writeFloat(p1.getPrecio());
escritor.writeFloat(p1.getUnidadesEnExistencia());

escritor.writeUTF(p2.getNombre());
escritor.writeFloat(p2.getPrecio());
escritor.writeFloat(p2.getUnidadesEnExistencia());

escritor.writeUTF(p3.getNombre());
escritor.writeFloat(p3.getPrecio());
escritor.writeFloat(p3.getUnidadesEnExistencia());
```

Recuerda que la clase *DataOutputStream* proporciona un *buffer* que permite escribir datos de tipo primitivo en un fichero (*int*, *float*, *double*, etc.) para lo cual utilizamos los siguientes métodos:

- ***writeBoolean(boolean valor)***: escribe un valor de tipo *boolean* en el fichero.
- ***writeByte(byte valor)***: escribe un valor de tipo *byte* en el fichero.
- ***writeBytes(String cadena)***: escribe cada uno de los bytes que forman parte de la cadena especificada en el argumento.
- ***writeChar(int valor)***: escribe el carácter asociado al código pasado como argumento.
- ***writeChars(String cadena)***: escribe cada uno de los caracteres de la cadena.

- **`writeDouble(double valor)`**: escribe un valor de tipo *double*.
- **`writeFloat(float valor)`**: escribe un valor de tipo *float*.
- **`writeInt(int valor)`**: escribe un valor de tipo *int*.
- **`writeLong(long valor)`**: escribe un valor de tipo *long*.
- **`writeShort(int valor)`**: escribe el valor del argumento y lo almacena como un *short*.
- **`writeUTF(String cadena)`**: escribe la cadena en formato UTF.

Cerramos los dos flujos de datos (filtro e iniciador) cerrando de esta forma el fichero.

```
escritor.close();
fichero.close();
```

Si ya has ejecutado el programa, comprueba que existe el fichero *almacen.dat* en la carpeta del proyecto. Si lo abres con el bloc de notas comprobarás que no se trata de un archivo de texto, no se puede interpretar a simple vista, se trata de un archivo binario y necesitamos de un programa específico para leerlo.

Lectura de un fichero binario

En este apartado abriremos el fichero *almacen.dat* para lectura y mostraremos en pantalla los registros que contiene, es decir, los artículos del almacén.

Realizaremos los siguientes pasos:

1. Crear un objeto de la clase *FileInputStream* dejando abierto el fichero *almacen.dat* para lectura.
2. Crear un objeto de la clase *DataInputStream* (filtro) asociado al objeto *FileInputStream* (iniciador) para obtener un *buffer* que permita optimizar la lectura, proporcionando métodos que permitan leer datos elementales de tipo *int*, *float*, *double*, etc.
3. Leer datos secuencialmente hasta llegar al final del fichero.
4. Cerrar los objetos *DataInputStream* y *FileInputStream*, dejando así cerrado el fichero.

Puedes crear otro proyecto Eclipse para realizar la lectura, pero recuerda que necesitarás copiar la clase *Producto*.

Para leer datos de un fichero binario tienes que saber previamente la estructura que tiene dicho fichero. En nuestro caso no hay duda, sabemos que hemos guardado registros con la estructura *String, float, float* y justo así es como iremos recuperándolos.

```
import java.io.DataInputStream;
import java.io.EOFException;
```

```
import java.io.FileInputStream;
import java.io.IOException;

public class Principal {

    public static void main(String[] args) {
        FileInputStream fichero;
        DataInputStream lector;
        try {
            fichero = new FileInputStream("almacen.dat");
            lector = new DataInputStream (fichero);
        } catch (IOException e) {
            System.out.println("Ha ocurrido un error al abrir el
fichero");
            System.out.println(e.getMessage());
            return;
        }

        boolean eof = false;
        while (!eof) {
            try {
                String pro = lector.readUTF();
                float pre = lector.readFloat();
                float uni = lector.readFloat();
                Producto p = new Producto(pro, pre, uni);
                System.out.println(p);
            } catch (EOFException e1) {
                eof = true;
            } catch (IOException e2) {
                System.out.println("Ha ocurrido un error al leer los
registros");
                System.out.println(e2.getMessage());
                break; // sale del bucle while
            }
        }

        try {
            lector.close();
            fichero.close();
        } catch (IOException e) {
            System.out.println("Ha ocurrido un error al cerrar el
fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Si observas el programa anterior comprobarás que tiene la típica estructura a la que ya te has acostumbrado: apertura del fichero, lectura, cierre del fichero.

Lectura de datos con DataInputStream

DataInputStream provee métodos para la lectura secuencial de tipos de datos elementales desde un fichero binario:

- ***readBoolean()*:** lee un valor de tipo *boolean* del fichero.
- ***readByte()*:** lee un valor de tipo *byte* del fichero.
- ***readChar()*:** lee un valor de tipo *char* del fichero.
- ***readDouble()*:** lee un valor de tipo *double* del fichero.
- ***readFloat()*:** lee un valor de tipo *float* del fichero.
- ***readInt()*:** lee un valor de tipo *int* del fichero.
- ***readLong()*:** lee un valor de tipo *long* del fichero.
- ***readShort()*:** lee un valor de tipo *short* del fichero.
- ***readUTF()*:** lee una cadena en formato *UTF* del fichero.

Todos estos métodos desencadenan una excepción de tipo *EOFException* si es final de fichero y no hay más datos que leer.

Observa la parte del programa donde estamos realizando la lectura secuencial de cada uno de los registros:

```
boolean eof = false;
while (!eof) {
    try {
        String pro = lector.readUTF();
        float pre = lector.readFloat();
        float uni = lector.readFloat();
        Producto p = new Producto(pro, pre, uni);
        System.out.println(p);
    } catch (EOFException e1) {
        eof = true;
    } catch (IOException e2) {
        System.out.println("Ha ocurrido un error al leer los registros");
        System.out.println(e2.getMessage());
        break; // sale del bucle while
    }
}
```

Sabemos la estructura de cada artículo y en ese orden estamos leyendo (*UTF String, float, float*). Por otro lado, declaramos la variable *eof* con el valor *false* y posteriormente le asignaremos el valor *true* cuando se produzca una excepción de tipo *EOFException*. Esta nos está permitiendo la lectura secuencial con el bucle *while* mientras *eof* no tenga el valor *true*.

A parte de desencadenarse la excepción *EOFException* al llegar a fin de fichero, podría ocurrir algún otro tipo de error inesperado. Por esa razón hemos añadido un segundo bloque *catch*, que capturará otro tipo de *IOException*.

Lectura de teclado con InputStreamReader y BufferedReader

Hasta ahora hemos utilizado flujos de datos para leer o escribir ficheros, pero no hay que olvidar que los flujos de datos también nos permiten comunicarnos con otros dispositivos, por ejemplo, el teclado.

En este apartado utilizaremos la clase *BufferedReader* para realizar la lectura de datos desde teclado, pero antes vamos a realizar una pequeña introducción a la ya conocida clase ***System***.

La clase *System*

Ya te has familiarizado con la clase *System* a base de incluir la sentencia *System.out.println(...)* para escribir datos en pantalla, pero nunca nos hemos parado a analizarla un poco más detenidamente. Pues bien, la clase *System*, perteneciente al paquete *java.lang*, ofrece una serie de flujos de comunicación estándares para entrada / salida. *System.in*, *System.out* y *System.err*.

- ***System.in***: flujo de datos que nos permite la entrada de datos desde el teclado.
- ***System.out***: flujo de datos que nos permite la salida de datos a pantalla.
- ***System.err***: tiene la misma función que *System.out*, usándose generalmente para la presentación de mensajes de error.

System.out dispone de los métodos ***println(...)*** y ***print(...)*** que ya has utilizado en muchas ocasiones y que nos permiten la escritura de información en pantalla.

System.in nos abre una vía de comunicación en el teclado. En el próximo ejemplo vamos a ver cómo utilizarlo para permitir al usuario introducir su nombre por teclado utilizando un objeto *BufferedReader* como *buffer* para recibir dicho dato.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Principal {

    public static void main(String[] args) throws IOException {
        InputStreamReader teclado = new InputStreamReader(System.in);
        BufferedReader lectorTeclado = new BufferedReader(teclado);
```

```
        String nombre;
        System.out.print("Escribe aquí tu nombre: ");
        nombre = lectorTeclado.readLine();
        System.out.println("Hola "+nombre+", ¿cómo te va?");
        lectorTeclado.close();
        teclado.close();
    }

}
```

Como has podido comprobar, este programa no difiere mucho de los anteriores con sus tres pasos de apertura, lectura y cierre. La diferencia es que, en lugar de abrir un fichero, hemos abierto una vía de comunicación con un dispositivo; el teclado.

Gestión de excepciones en la lectura / escritura

Todas las excepciones que se producen en las operaciones de lectura / escritura son derivadas de *IOException*, clase base situada en *java.io*, al igual que el resto de las clases que hemos ido estudiando en esta unidad.

Recuerda el apartado anterior, cuando recorrimos secuencialmente los artículos del almacén. Retomemos ahora ese ejemplo para ver cómo se gestionaron las excepciones.

```
while (!eof) {
    try {
        String pro = lector.readUTF();
        float pre = lector.readFloat();
        float uni = lector.readFloat();
        Producto p = new Producto(pro, pre, uni);
        System.out.println(p);
    } catch (EOFException e1) {
        eof = true;
    } catch (IOException e2) {
        System.out.println("Ha ocurrido un error al leer los registros");
        System.out.println(e2.getMessage());
        break; // sale del bucle while
    }
}
```

Sabemos que se producirá una excepción de tipo *EOFException* al llegar al final de fichero, es decir, cuando se intente realizar una lectura de datos y ya no haya nada más que leer. Por ese motivo encerramos toda la operación de lectura / escritura en un *try* y capturamos en el primer *catch* la excepción *EOFException*. En otro *catch* capturamos la genérica *IOException* para que recoja cualquier otra excepción que pueda producirse. Recuerda que el orden es muy importante, ya que una *EOFException* también es una *IOException* y si intercambiamos el orden nunca llegará a ejecutarse el bloque del segundo *catch*.

Las excepciones que más utilizarás en las operaciones de lectura / escritura de fichero son: *FileNotFoundException* (fichero no encontrado) y *EOFException* (final de fichero).

Clase *IOException*: pulsa [aquí](#) para acceder a la página oficial de Oracle sobre la clase *IOException*.

La clase scanner

Introducción a la clase Scanner

La clase *Scanner* apareció con la versión 5 de Java para simplificar enormemente tareas de entrada o lectura de datos.

Como su nombre indica, actúa como un lector exclusivamente, no puede realizar operaciones de escritura.

Podemos leer con un objeto de la clase *Scanner* desde diversas fuentes y el origen de los datos se especifica a través del parámetro del constructor de la clase de *Scanner*.

Con un objeto de la clase *Scanner* podemos leer:

- **Desde un String.**

String texto = "Este es el texto que será leído por un objeto Scanner";

Scanner lector = new Scanner(texto);

El origen de la lectura es un *String*.

- **Desde el teclado.**

Scanner lector = new Scanner(System.in);

El origen de la lectura es el teclado.

- **Desde un fichero de texto.**

Scanner lector = new Scanner(new File("peliculas.txt"));

El origen de la lectura es un archivo.

Scanner fragmenta los datos a leer y los va recuperando por medio de los métodos *next()*, *nextLine()*, *nextInt()*, *nextFloat()*, etc. En los siguientes apartados irás descubriendo aplicaciones prácticas del uso de la clase *Scanner*.

Lectura de cadenas de texto

La clase *Scanner* permite fragmentar un *String* y leer secuencialmente cada uno de los fragmentos.

Para dividir el *String* en fragmentos se guía por una subcadena que actúa como separador. Si no se especifica lo contrario el separador será el espacio en blanco, de modo que la lectura por defecto será palabra a palabra.

Vamos a poner en práctica varios ejemplos de lectura secuencial de un *String* con un objeto *Scanner*.

Leer un *String* palabra a palabra

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto = "La cripta mágica";
        Scanner lector = new Scanner(texto);
        while (lector.hasNext()) {
            System.out.println(lector.next());
        }
        lector.close();
    }
}
```

Scanner proporciona un flujo de lectura, cuyo origen se especifica en el constructor. En este caso le hemos proporcionado una cadena con el texto "La cripta mágica", que será fragmentada por palabras. El método *lector.next()* devuelve el siguiente fragmento de la cadena de manera secuencial, por esa razón lo encerramos dentro de un *while* con la condición *lector.hasNext()*, ya que *hasNext()* devuelve *true* mientras existan más fragmentos.

Como puedes observar, volvemos a utilizar la habitual secuencia de acciones: apertura del flujo, lectura, cierre.

Leer un *String* línea a línea

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="Frodo Bolsón\nSamsagaz Gamyi\nPeregrin Tuk\nMeriadoc
Brandigamo";
        Scanner lector = new Scanner(texto);
        while (lector.hasNext()) {
            System.out.println(lector.nextLine());
        }
        lector.close();
    }
}
```

El método *nextLine()* lee una línea completa sin fragmentarla. En la cadena *texto* hemos colocado tres veces el carácter de escape "\n" para insertar retornos de carro, generando así cuatro líneas.

Leer un *String* según separador personalizado

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="Rosa López;Miguel de la Parra;Carmen Ruiz;Francisco
López;Rosa Morales";
        Scanner lector = new Scanner(texto);
        lector.useDelimiter(";");
        while (lector.hasNext()){
            System.out.println(lector.next());
        }
        lector.close();
    }
}
```

Con la sentencia `lector.useDelimiter(";")`; hemos establecido el delimitador utilizado para fragmentar la cadena, de modo que habrá cuatro fragmentos para leer.

Lectura de un *String* con datos numéricos

Si la cadena que deseamos fragmentar contiene datos numéricos, es posible que nos interese recuperar los datos no como textos, sino como datos de tipo `int`, `float`, `double`, etc.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="80;25;48;56;38;46";
        Scanner lector = new Scanner(texto);
        lector.useDelimiter(";");
        while (lector.hasNext()){
            int num = lector.nextInt();
            System.out.println(num);
        }
        lector.close();
    }
}
```

En este ejemplo, la cadena contiene cantidades numéricas enteras separadas por punto y coma.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String texto="Tomates;3;1,5;Patatas;5;3,5;Pimientos;1;0,95";
        Scanner lector = new Scanner(texto);
```

```
lector.useDelimiter(";");
while (lector.hasNext()) {
    String producto = lector.next();
    int cantidad = lector.nextInt();
    float precio = lector.nextFloat();
    float total = cantidad*precio;
    System.out.println(cantidad + " kg de " + producto + " a " +
precio + "€ = " + total);
}
lector.close();
}
```

En este ejemplo, la cadena contiene datos de ventas y hay que ir leyendo bloques de tres elementos formatos por producto (*String*), cantidad (*int*) y precio (*float*).

IMPORTANTE: *nextInt()*, *nextFloat()*, *nextDouble()*, etc. podrían provocar una excepción de tipo *InputMismatchException* si el elemento leído no tiene el formato correcto para poder ser convertido al tipo numérico que corresponda.

Lectura desde teclado

En este apartado profundizaremos sobre la entrada de datos por teclado con la clase *Scanner*.

Scanner dispone de varios métodos para recibir datos desde el teclado. Comenzaremos por recordar el sistema más sencillo que ya hemos empleado en otras ocasiones, el método ***nextLine()***, que recibe una línea de texto finalizada con un retorno de carro.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        String nombre;
        Scanner lector = new Scanner (System.in);
        System.out.println("¿Cómo te llamas?");
        nombre = lector.nextLine();
        System.out.println("Encantado " + nombre);
        lector.close();
    }
}
```

El método *nextLine()* lee una entrada por teclado finalizada con un retorno de carro, es decir, una línea de texto. Guardamos la línea leída en la variable *nombre*.

También podemos recibir por teclado datos numéricos utilizando los métodos *nextInt()*, *nextFloat()*, *nextDouble()*, etc.

Veamos un ejemplo en el que deseamos preguntar al usuario primero su edad, para lo que utilizaremos el método `nextInt()` y después su nombre, para lo que utilizaremos el método `nextLine()`.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        Scanner lector = new Scanner(System.in);
        System.out.println("¿Cuántos años tienes?");
        int edad = lector.nextInt();
        System.out.println("¿Cómo te llamas?");
        String nombre = lector.nextLine();
        System.out.println("Encantado " + nombre + " de " + edad + " años");
        lector.close();
    }
}
```

Programa que lee desde el teclado un *int* y una línea de texto.

El programa parece sencillo, sin embargo, no funciona como esperamos.

```
¿Cuántos años tienes?  
18  
¿Cómo te llamas?  
Encantado de 18 años  
Pide la edad y directamente, sin darnos opción a introducir el nombre, pasa a ejecutar la siguiente línea mostrando en pantalla "Encantado de 18 años".
```

¿Y por qué ocurre esto?

Durante la ejecución, al escribir la edad, no solo introdujiste un número, sino que también **pulsaste la tecla "enter"** y ahí es donde está el quid de la cuestión. La sentencia `edad = lector.nextInt()` leyó el número, pero el retorno de carro producido por la tecla "enter" quedó en el *buffer* del teclado y fue recogido por la sentencia `String nombre = lector.nextLine()`, quedando la entrada del nombre para una tercera lectura, que nunca se produjo.

Una posible solución es sustituir `nextLine()` por `next()`, pero en ese caso tendríamos problemas para recibir nombres compuestos, ya que cada lectura corresponde a una palabra.

Otra solución está en colocar un `nextLine()` detrás de cada `nextInt()`, `nextFloat()`, `nextDouble()`, etc. para que recoja el retorno de carro.

Esta última versión resuelve el problema:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
```

```
Scanner lector = new Scanner(System.in);
System.out.println("¿Cuántos años tienes?");
int edad = lector.nextInt(); // Recibe un int
lector.nextLine(); // Recibe el retorno de carro.
System.out.println("¿Cómo te llamas?");
String nombre = lector.nextLine(); // Recibe un string
System.out.println("Encantado " + nombre + " de " + edad + " años");
lector.close();
}
```

Lectura de ficheros de texto

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) throws FileNotFoundException {
        File fichero = new File("C:\\cine\\peliculas.txt");
        if (!fichero.exists()) { // Si no existe el fichero
            System.out.println("El fichero no existe");
            return;
        }

        Scanner lector = new Scanner(fichero);
        while (lector.hasNext()) {
            String linea = lector.nextLine();
            System.out.println(linea);
        }

        lector.close();
    }
}
```

Ejecutamos sucesivas veces el método *nextLine()* mientras se cumpla la condición *lector.hasNext()*, es decir, mientras sigan quedando líneas por leer.

Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Utilizamos la clase ***File*** para obtener **información sobre archivos y directorios**, así como crearlos, renombrarlos o eliminarlos.
- Todas las clases que representan **flujos de caracteres** en formato Unicode de 16 bits derivan directa o indirectamente de las clases abstractas ***Reader*** y ***Writer***.
- Todas las clases que representan **flujos de bytes** (información binaria) derivan directa o indirectamente de las clases abstractas ***InputStream*** y ***OutputStream***.
- La clase ***Scanner***, disponible a partir de la versión 5 de Java, actúa como **flujo de lectura de caracteres** desde un objeto *String*, el teclado o un archivo de texto.
- ***IOException*** es la clase base de la que derivan todas las demás clases de excepción que tienen que ver con operaciones de entrada / salida de datos.

1.2. Lectura / escritura de objetos



Índice

Introducción a la unidad formativa	4
Objetivos.....	4
Lectura y escritura de objetos.....	5
La interfaz Serializable	5
Grabar un objeto en disco	8
Recuperar un objeto de disco.....	11
El modificador transient	12
Grabar y recuperar varios objetos.....	12
Despedida	17
Resumen.....	17

El contenido de esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.

Introducción a la unidad formativa

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

- Crear clases que implementan la interfaz *Serializable*.
- Guardar objetos en disco utilizando la clase *ObjectOutputStream* como flujo de escritura de objetos.
- Recuperar objetos guardados en disco utilizando la clase *ObjectInputStream* como flujo de lectura de objetos.

¡Ánimo y adelante!

Lectura y escritura de objetos

La interfaz Serializable

La interfaz **Serializable** aporta a las clases que la implementan la capacidad de persistencia de sus objetos.

¿Y qué es la persistencia?

Según el diccionario de la Real Academia Española, persistir significa "durar a lo largo del tiempo".

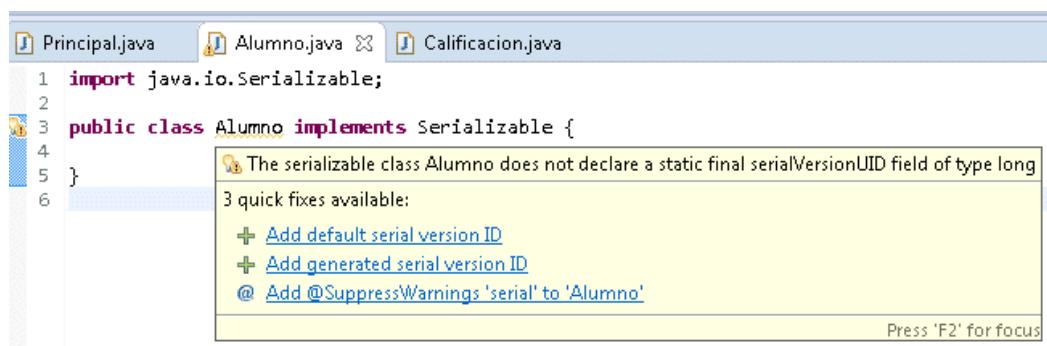
Hasta ahora, cuando creamos un objeto en un programa, el objeto solo dura lo que dura la ejecución de dicho programa, incluso menos, ya que si el objeto se ha declarado dentro de un método, su duración será el tiempo de ejecución del método, no del programa.

Si una clase implementa la **interfaz Serializable**, los objetos que pertenecen a dicha clase podrán ser grabados en disco y recuperados las veces que sea necesario, incluso en distintos programas.

Vamos a crear una clase llamada *Alumno* con capacidad de persistencia. Para ello crea un nuevo proyecto en Eclipse con el nombre que deseas y comienza por crear la clase *Alumno*, de momento vacía como ves a continuación:

```
import java.io.Serializable;
public class Alumno implements Serializable { }
```

Si ya has creado la clase, comprobarás que Eclipse te está dando una alerta, la palabra *Alumno* está subrayada en amarillo y, si sitúas el puntero de ratón durante un rato sobre la palabra *Alumno*, obtendrás un cuadro con información sobre la alerta y las soluciones propuestas por Eclipse.



La alerta nos dice lo siguiente: "*The serializable class Alumno does not declare a static final serialVersionUID fields of type long*". Está claro que nos está pidiendo que declaremos una variable llamada *serialVersionUID*.

¿Qué es serialVersionUID?

La serialVersionUID es el número de versión de la clase, y se utiliza para evitar problemas de incompatibilidad de versión en los procesos de serialización y deserialización entre los programas que hacen de emisor y receptor del objeto. ¿Y cuándo se produce un problema de incompatibilidad? Lo comprenderás mejor con un ejemplo:

- Tenemos un programa *A* con una clase *Alumno* que cuenta con las propiedades *nombre*, *edad* y *telefono*. Dentro de la clase *Principal* creamos un objeto *Alumno* y lo guardamos en un archivo llamado *datos.dat*. El programa *A* es el que realiza la serialización y por lo tanto el emisor del objeto guardado.
- Tenemos otro programa *B*, donde hemos copiado la clase *Alumno*, pero esta vez se nos ha ocurrido añadir una propiedad más llamada *domicilio*. Los programas *A* y *B* tienen distinta versión de la clase *Alumno*.

Ahora desde la clase *Principal* del programa *B* recuperamos el objeto *Alumno* que previamente guardamos en el archivo *datos.dat* durante la ejecución del programa *A*. El programa *B* debe realizar la deserialización del objeto guardado y por lo tanto será el receptor de dicho objeto.

El programa *B* nos arroja un excepción, ya que intenta recuperar un objeto construido a partir de la primera versión de la clase *Alumno*, sin embargo, el programa *B* contiene la segunda versión de la clase *Alumno*, que resulta incompatible.

Las versiones primera y segunda de la clase *Alumno* deberían tener distinto *serialVersionUID* para poder distinguir rápidamente que se trata de versiones distintas de la misma clase. De esta forma, en el proceso de deserialización, la máquina virtual de Java comparará la *serialVersionUID* del objeto guardado con la *serialVersionUID* de la clase *Alumno* que contiene el programa *B*, arrojando una excepción de tipo ***InvalidClassException***.

¿Y qué pasa si no declaramos la serialVersionUID?

Si dentro de la clase no hemos especificado un valor de *serialVersionUID*, la máquina virtual de Java debe examinar las clases de origen y destino generando las *serialVersionUID* de forma dinámica. Aunque no sea obligatorio, resulta mucho más rápido y efectivo declarar la *serialVersionUID* y modificarla en cada nueva versión.

Volvamos a poner de nuevo la atención en la clase Alumno.

Vuelve a situar el puntero del ratón sobre el nombre de la clase hasta que salga el cuadro informativo y selecciona la opción "*add generated serial version ID*". Verás que se añade automáticamente la variable *serialVersionUID* con un valor auto calculado.

```
import java.io.Serializable;

public class Alumno implements Serializable {
    private static final long serialVersionUID = 4854486451470258537L;

}
```

Verás que si borras la nueva línea para que vuelva a salir la alerta y vuelves a realizar de nuevo la operación, generará exactamente el mismo número. Sin embargo, si modificas algo, por ejemplo, añadiendo dos propiedades, generará distinto número. Cambia ahora la clase *Alumno* dejándola así:

```
import java.io.Serializable;

public class Alumno implements Serializable {

    private String nombre;
    private int edad;

}
```

Ahora vuelve a realizar la operación anterior para que vuelva a generarse la *serialVersionUID*.

```
import java.io.Serializable;

public class Alumno implements Serializable {

    private static final long serialVersionUID = 1742837368213302555L;
    private String nombre;
    private int edad;

}
```

El número de versión se genera a partir de un complejo algoritmo que tiene que ver con el contenido de la clase y es susceptible a los cambios que realicemos dentro del código.

En los siguientes apartados completaremos la clase *Alumno* y comenzaremos a realizar operaciones de lectura y escritura de objetos.

Grabar un objeto en disco

En este apartado veremos cómo grabar un objeto de la clase *Alumno* en un archivo.

Para empezar, debes completar el código de la clase *Alumno* copiando y pegando el siguiente código:

```
import java.io.Serializable;
import java.util.ArrayList;

public class Alumno implements Serializable {
    private static final long serialVersionUID = 4854486451470258537L;

    private String nombre;
    private int edad;
    private ArrayList<Calificacion> calificaciones;

    public Alumno(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        this.calificaciones = new ArrayList<Calificacion>();
    }

    public void calificar(String asignatura, int nota) {
        this.calificaciones.add(new Calificacion(asignatura, nota));
    }

    public String getNombre() {
        return nombre;
    }
    public int getEdad() {
        return edad;
    }
    public ArrayList<Calificacion> getCalificaciones() {
        return calificaciones;
    }
}
```

Como puedes comprobar por el código, un objeto *Alumno* está formado por las propiedades *nombre*, *edad* y una colección de objetos *Calificacion*.

La clase *Calificacion* tiene la siguiente implementación:

```
import java.io.Serializable;

public class Calificacion implements Serializable {
    private static final long serialVersionUID = 3057545624874202352L;

    private String asignatura;
    private int nota; // Sobre 100
```

```
public Calificacion(String asignatura, int nota) {  
    this.asignatura = asignatura;  
    this.nota = nota;  
}  
  
@Override  
public String toString() {  
    return "Calificación [Asignatura=" + asignatura + ", Nota=" + nota +  
"]";  
}  
}
```

Ya tenemos todo listo para construir un objeto *Alumno* y persistirlo grabándolo en un archivo.

```
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.ObjectOutputStream;  
  
public class Principal {  
    public static void main(String args[]) {  
        // Crear objeto Alumno  
        Alumno alu1 = new Alumno("Pedro", 25);  
        alu1.calificar("Matemáticas", 50);  
        alu1.calificar("Inglés", 75);  
        alu1.calificar("Informática", 95);  
        alu1.calificar("Lengua", 60);  
  
        // Abrir fichero para escritura  
        FileOutputStream file;  
        ObjectOutputStream buffer;  
        try {  
            file = new FileOutputStream("J:\\alumno.dat");  
            buffer = new ObjectOutputStream(file);  
        } catch (IOException e) {  
            System.out.println("No se ha podido abrir el fichero");  
            System.out.println(e.getMessage());  
            return;  
        }  
  
        // Guarda objeto en el fichero alumno.dat  
        try {  
            buffer.writeObject(alu1);  
            System.out.println("El objeto se ha grabado con éxito");  
        } catch (IOException e) {  
            System.out.println("Error al escribir en el fichero");  
            System.out.println(e.getMessage());  
        }  
  
        // Cerrar el fichero  
        try {  
            buffer.close();  
            file.close();  
        } catch (IOException e) {  
        }  
    }  
}
```

```
        System.out.println("Error al cerrar el fichero");
        System.out.println(e.getMessage());
    }
}
```

Antes de ejecutar el programa vamos a analizarlo un poco.

- En primer lugar hemos creado un objeto de la clase *Alumno* llamado *alu1* y le hemos añadido cuatro calificaciones.
- **file = new FileOutputStream("J:\\alumno.dat");
buffer = new ObjectOutputStream(file);**
Después hemos construido un objeto de la clase *FileOutputStream* (iniciador) dejando el archivo *alumno.dat* abierto para escritura. El fichero será creado en cada ejecución, sobrescribiendo la versión anterior si existe. También podemos abrir el archivo para añadir sin eliminar los datos anteriores; en ese caso tendríamos que establecer a *true* el segundo argumento: *file = new FileOutputStream("J:\\\\alumno.dat", true);*

En el ejemplo, el archivo se guardará en una memoria USB que se encuentra en la unidad J. Cambia la ruta con la ubicación que deseas.

Ya abierto el fichero, creamos un objeto *ObjectOutputStream* que nos servirá como filtro para mejorar el proceso de escritura.

- **buffer.writeObject(alu1);**
El método *writeObject()* de la clase *ObjectOutputStream* es el que nos permite grabar el objeto en disco.
- **buffer.close();
file.close();**
Por último, igual que en todas las operaciones de entrada / salida, hay que terminar cerrando los flujos de datos, liberando así el fichero.

Ahora ya puedes ejecutar el programa.

Si todo ha salido bien, habrás obtenido en pantalla el mensaje "El objeto se ha grabado con éxito". Comprueba con ayuda del explorador de Windows si el archivo se ha creado.

Recuperar un objeto de disco

Ha llegado el momento de recuperar el objeto *Alumno* guardado en disco.

Comienza por crear un nuevo proyecto, pero necesitarás la implementación de las clases *Alumno* y *Calificacion*, así que puedes copiarlas del proyecto anterior.

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) throws ClassNotFoundException {

        // Abrir fichero para lectura
        FileInputStream file;
        ObjectInputStream buffer;
        try {
            file = new FileInputStream("J:\\alumno.dat");
            buffer = new ObjectInputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Lee el objeto guardado en el archivo alumno.dat
        try {
            Alumno alu1 = (Alumno) buffer.readObject();
            System.out.println("Nombre del alumno: " + alu1.getNombre());
            System.out.println("Edad: " + alu1.getEdad());
            for (Calificacion c : alu1.getCalificaciones()) {
                System.out.println(c);
            }
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Como has podido comprobar, hemos seguidos los tres pasos habituales: abrir, leer, cerrar. Esta vez hemos utilizado los flujos de lectura *FileInputStream* (iniciador) y *ObjectInputStream* (filtro).

Con la sentencia

```
Alumno alu1 = (Alumno) buffer.readObject();
```

hemos leído el objeto, pero hemos tenido que convertirlo a tipo *Alumno*, ya que el método *readObject()* devuelve un genérico *Object*.

El modificador transient

El modificador *transient* se utiliza con clases serializables para indicar las propiedades que no queremos que sean serializadas, es decir, las que no deseamos que se guarden. Tiene sentido con algunas propiedades, tales como un *password*.

Para ponerlo en práctica sigue estos pasos

1. Añade el modificador *transient* a la propiedad *edad* de la clase *Alumno*.

```
private transient int edad;
```

2. Vuelve a ejecutar el programa creado en el apartado "Grabar un objeto en disco". A pesar de que construimos un objeto *Alumno* con *nombre* "Pedro" y *edad* 25 años, la *edad* no se habrá guardado.
3. Ahora vuelve a ejecutar el programa creado en el apartado "Recuperar un objeto de disco". El resultado es el siguiente:

Nombre del alumno: Pedro

Edad: 0

Calificación [Asignatura=Matemáticas, Nota=50]

Calificación [Asignatura=Inglés, Nota=75]

Calificación [Asignatura=Informática, Nota=95]

Calificación [Asignatura=Lengua, Nota=60]

Como la *edad* no se guardó, el objeto recuperado tiene el valor por defecto para una variable de tipo *int*, es decir, un cero.

Grabar y recuperar varios objetos

En este apartado veremos cómo podemos guardar en un fichero varios objetos del mismo tipo cómo podemos posteriormente leerlos controlando el final del fichero.

Como ejemplo, vamos a crear una agenda de contactos. En primer lugar debes crear un nuevo proyecto en Eclipse y una clase *Contacto*, que representará a cada uno de los contactos que queremos guardar en la agenda.

```
import java.io.Serializable;

public class Contacto implements Serializable {
    private static final long serialVersionUID = -4624046047796483183L;

    private String nombre;
    private String telefono;

    public Contacto(String nombre, String telefono) {
        this.nombre = nombre;
        this.telefono = telefono;
    }

    public String getNombre() {
        return nombre;
    }
    public String getTelefono() {
        return telefono;
    }

    @Override
    public String toString() {
        return "Contacto [" + nombre + " - " + telefono + "]";
    }
}
```

Ahora, en la clase *Principal*, guardarás tres contactos en el fichero *agenda.dat*.

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

public class Principal {

    public static void main(String[] args) {
        // Abrimos fichero para escritura
        FileOutputStream file;
        ObjectOutputStream buffer;
        try {
            file = new FileOutputStream("D:\\agenda.dat", true);
            buffer = new ObjectOutputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Creamos tres contactos
```

```
    Contacto c1 = new Contacto("Amelia", "913670542");
    Contacto c2 = new Contacto("Federico", "6166644422");
    Contacto c3 = new Contacto("Carmen", "639888777");

    // Guardamos los tres contactos en agenda.dat
    try {
        buffer.writeObject(c1);
        buffer.writeObject(c2);
        buffer.writeObject(c3);
        System.out.println("Los contactos se han guardado con éxito");
    } catch (IOException e) {
        System.out.println("Error al escribir en el fichero");
        System.out.println(e.getMessage());
    }

    // Cerrar el fichero
    try {
        buffer.close();
        file.close();
    } catch (IOException e) {
        System.out.println("Error al cerrar el fichero");
        System.out.println(e.getMessage());
    }
}
```

Leer contactos hasta que sea final de fichero

Ahora vamos a realizar un listado de contactos, para lo cual debes crear un nuevo proyecto con el nombre que deseas y copiar la clase *Contacto* del proyecto anterior. Para leer cada contacto debes utilizar el método *readObject()*, tal como ya has aprendido, pero en esta ocasión hay varios objetos de la clase *Contacto* para leer dentro del fichero, con lo cual necesitarás iterar, es decir, utilizar un bucle para leer sucesivas veces mientras no sea final de fichero.

¿Pero cómo podemos saber cuándo se ha llegado al final de fichero?

La solución está en la excepción *EOFException*

Cuando intentamos leer un objeto del flujo de lectura y ya no hay más objetos para leer se produce una excepción de tipo *EOFException*. En el programa que sigue estamos capturando la excepción *EOFException* cuando se produce y realizando la siguiente asignación: *eof = true*. De esta forma podemos leer objetos mientras la variable *eof* se mantenga con el valor *false*.

```
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;

public class Principal {
```

```
public static void main(String args[])  {

    // Abrimos fichero agenda.dat para lectura
    FileInputStream file;
    ObjectInputStream buffer;
    try {
        file = new FileInputStream("D:\\agenda.dat");
        buffer = new ObjectInputStream(file);
    } catch (IOException e) {
        System.out.println("No se ha podido abrir la agenda de
contactos");
        System.out.println(e.getMessage());
        return;
    }

    // Leemos la lista de contactos
    boolean eof = false;
    Contacto c;
    while (!eof) {
        try {
            c = (Contacto) buffer.readObject();
            System.out.println(c);
        } catch (EOFException e1) {
            eof = true;
        } catch (IOException e2) {
            System.out.println("Error al leer los contactos de la
agenda");
            System.out.println(e2.getMessage());
        } catch (ClassNotFoundException e3) {
            System.out.println("La clase Contacto no está cargada en
memoria");
            System.out.println(e3.getMessage());
        }
    }

    // Cerramos el fichero
    try {
        buffer.close();
        file.close();
    } catch (IOException e) {
        System.out.println("Error al cerrar el fichero");
        System.out.println(e.getMessage());
    }
}

}
```

Controlar el final de fichero con el método *available()*

Otro sistema para controlar cuándo hemos llegado a final de fichero es mediante el método *available()* de la clase *FileInputStream*, que devuelve un número entero con el número de bytes pendientes de lectura.

En el siguiente programa estamos asignando a la variable bytesEnBuffer el valor devuelto por el método available() antes de comenzar la lectura y después de leer cada objeto. De esta forma podemos leer mientras se cumpla la siguiente condición: bytesEnBuffer>0.

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;

public class Principal {
    public static void main(String args[]) throws IOException {
        int bytesEnBuffer;

        // Abrimos fichero agenda.dat para lectura
        FileInputStream file;
        ObjectInputStream buffer;
        try {
            file = new FileInputStream("D:\\agenda.dat");
            buffer = new ObjectInputStream(file);
            bytesEnBuffer = file.available();
        } catch (IOException e) {
            System.out.println("No se ha podido abrir la agenda de contactos");
            System.out.println(e.getMessage());
            return;
        }

        // Leemos la lista de contactos
        System.out.println("Bytes por leer: " + bytesEnBuffer);
        Contacto c;
        while (bytesEnBuffer>0) {
            try {
                c = (Contacto) buffer.readObject();
                System.out.println(c);
            } catch (IOException e2) {
                System.out.println("Error al leer los contactos de la agenda");
                System.out.println(e2.getMessage());
            } catch (ClassNotFoundException e3) {
                System.out.println("La clase Contacto no está cargada en memoria");
                System.out.println(e3.getMessage());
            }
            bytesEnBuffer = file.available();
            System.out.println("Bytes pendientes en buffer: " +
bytesEnBuffer);
        }

        // Cerramos el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Para que un objeto tenga la capacidad de persistencia debe implementar la interfaz ***Serializable***.
- Podemos guardar objetos en disco utilizando la clase ***ObjectOutputStream***.
- Podemos recuperar objetos guardados en disco utilizando la clase ***ObjectInputStream***.

MP_0486. Acceso a datos

**UF2. Manejo de conectores a
bases de datos relacionales**

2.1. Introducción a las bases de datos



Índice

Introducción a la unidad formativa	4
Objetivos.....	4
Introducción a las bases de datos	5
Definición de base de datos.....	5
Modelos de base de datos.....	6
Estructura de una base de datos.....	6
Dominio de los campos o atributos.....	8
El lenguaje SQL	8
La sentencia SQL <i>INSERT</i>	8
La sentencia SQL <i>SELECT</i>	9
La sentencia SQL <i>UPDATE</i>	9
La sentencia SQL <i>DELETE</i>	9
Los sublenguajes de SQL.....	10
Despedida	11
Resumen.....	11

El contenido de esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.

Introducción a la unidad formativa

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

- Definir el término "base de datos".
- Enumerar y describir todos los conceptos asociados a las bases de datos: tablas, filas o registros, atributos o campos, dominio de los atributos, etc.

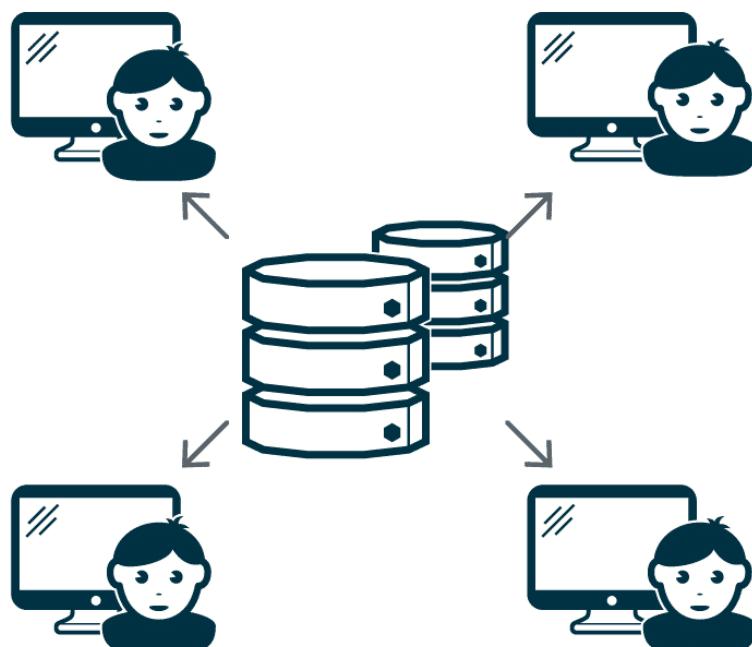
¡Ánimo y adelante!

Introducción a las bases de datos

Definición de base de datos

Una base de datos es un “almacén” que nos permite guardar grandes cantidades de información de forma organizada para que, posteriormente, podamos encontrarla y utilizarla fácilmente.

Una base de datos es gestionada por un programa gestor de bases de datos (**DBMS** - Data Base Management System o **SGBD** - Sistema Gestor de Base de Datos).



El DBMS garantiza que la base de datos esté disponible para los usuarios y programas que la requieran.

Algunos ejemplos de DBMS son:

- Microsoft SQL Server.
- MySQL.
- Oracle Database.
- PostgreSQL.
- Microsoft Access.
- IBM DB2.

Modelos de base de datos

Un modelo de datos determina la estructura lógica en la que una base de datos va a ser construida. Por lo tanto, determina de qué manera los datos son almacenados, organizados y manipulados. Cada DBMS trabaja con un modelo de base de datos determinado. Existen numerosos modelos que determinan la estructura lógica de una base de datos. Entre ellos:

- Modelo jerárquico.
- Modelo en red.
- Modelo de estrella.
- Modelo relacional.
- Modelo de base de datos orientada a objetos.

En esta lección estudiaremos el modelo relacional, donde una base de datos está formada por un conjunto de tablas o relaciones de datos.

Estructura de una base de datos

Una base de datos está formada por una o más tablas, donde cada tabla guarda una relación de entidades o elementos del mismo tipo, por ejemplo: clientes, facturas, productos, etc.

CLIENTES				
NIF	NOMBRE	DOMICILIO	TLF	CIUDAD
43434343-A	DELGADO PEREZ MARISA	C/MIRAMAR, 84 3ºA	925-200-967	TOLEDO
51592939-K	LOPEZ VAL SOLEDAD	C/PEZ, 54 4ºC	915-829-394	MADRID
51639989-K	DELGADO ROBLES MIGUEL	C/OCA, 54 5ºC	913-859-293	MADRID
51664372-R	GUTIERREZ PEREZ ROSA	C/CASTILLA, 4 4ºA	919-592-932	MADRID

Tabla que guarda una relación de clientes. Cada fila contiene la información de un cliente.

Características de una tabla

- Cada tabla se compone de **filas y columnas**.
- **Cada fila de una tabla conforma un registro** que guarda la información completa sobre una determinada entidad o elemento. En nuestro ejemplo de la imagen anterior cada registro contiene la información de un cliente.
- Las **columnas** guardan una información concreta sobre cada elemento: un nombre, un DNI, etc. A las columnas las denominamos **campos**.

CLIENTES				
NIF	NOMBRE	DOMICILIO	TLF	CIUDAD
43434343-A	DELGADO PEREZ MARISA	C/MIRAMAR, 84 3ºA	925-200-967	TOLEDO
51592939-K	LOPEZ VAL SOLEDAD	C/PEZ, 54 4ºC	915-829-394	MADRID
51639989-K	DELGADO ROBLES MIGUEL	C/OCA, 54 5ºC	913-859-293	MADRID
51664372-R	GUTIERREZ PEREZ ROSA	C/CASTILLA, 4 4ºA	919-592-932	MADRID

Fila o registro →

↑ Columna o campo

Cada tabla suele tener una clave principal, que identifica a cada uno de los registros. La clave principal no podrá tener valores duplicados.

CLIENTES				
NIF	NOMBRE	DOMICILIO	TLF	CIUDAD
43434343-A	DELGADO PEREZ MARISA	C/MIRAMAR, 84 3ºA	925-200-967	TOLEDO
51592939-K	LOPEZ VAL SOLEDAD	C/PEZ, 54 4ºC	915-829-394	MADRID
51639989-K	DELGADO ROBLES MIGUEL	C/OCA, 54 5ºC	913-859-293	MADRID
51664372-R	GUTIERREZ PEREZ ROSA	C/CASTILLA, 4 4ºA	919-592-932	MADRID

↑ El NIF sirve para identificar inequívocamente a cada cliente.

La clave principal de una tabla sirve además para establecer interrelaciones con otras tablas de la base de datos. Una tabla de facturas podría tener entre sus campos el NIF del cliente al que corresponde dicha factura, así se establece una asociación entre los clientes y sus facturas.

CLIENTES				
NIF	NOMBRE	DOMICILIO	TLF	CIUDAD
43434343-A	DELGADO PEREZ MARISA	C/MIRAMAR, 84 3ºA	925-200-967	TOLEDO
51592939-K	LOPEZ VAL SOLEDAD	C/PEZ, 54 4ºC	915-829-394	MADRID
51639989-K	DELGADO ROBLES MIGUEL	C/OCA, 54 5ºC	913-859-293	MADRID
51664372-R	GUTIERREZ PEREZ ROSA	C/CASTILLA, 4 4ºA	919-592-932	MADRID

FACTURAS				
NUMERO	FECHA	PAGADO	NIF	IMPORTE
5440	05-sep-11	Sí	43434343A	27,00 €
5441	05-sep-11	Sí	51639989K	53,00 €
5442	06-sep-11	No	43434343A	85,00 €
5443	10-oct-11	Sí	51639989K	43,00 €
5444	13-oct-11	Sí	51664372R	12,00 €
5445	14-oct-11	No	43434343A	18,00 €

Dominio de los campos o atributos

Como vimos en el apartado anterior, cada columna de una tabla se corresponde con un campo, también denominado atributo.

El dominio de un campo es el conjunto de valores posibles que puede tomar dicho campo.

Cuando se define el diseño de una tabla **hay que especificar el tipo de dato que corresponde a cada uno de los campos**. Además del tipo de dato, también pueden definirse restricciones. Por ejemplo, el tipo de dato del campo *IMPORTE* de la tabla *FACTURA* es numérico, y una restricción podría ser que no admita números negativos. El dominio del campo viene finalmente determinado por el tipo de datos y otras restricciones aplicadas.

En los siguientes apartados verás los tipos de datos que se pueden establecer en una base de datos relacional creada en MySQL. Si bien en otros DBMS pueden variar ligeramente.

El lenguaje SQL

SQL es un lenguaje estándar para almacenar, manipular y recuperar información de una base de datos. Al conjunto de operaciones que podemos realizar con el lenguaje SQL se le denomina **CRUD** abreviatura de:

1. **Create**: crear nuevos registros o filas. Se lleva a cabo con la sentencia ***INSERT***.
2. **Read**: leer filas. Se lleva a cabo con la sentencia ***SELECT***.
3. **Update**: modificar filas. Se lleva a cabo con la sentencia ***UPDATE***.
4. **Delete**: borrar filas. Se lleva a cabo con la sentencia ***DELETE***.

La sentencia SQL *INSERT*

La sentencia *INSERT* se utiliza para añadir nuevas filas o registros a una tabla. Tiene el siguiente formato:

INSERT INTO nombre_tabla[(campo_1,...,campo_m)] VALUES [(campo_1,...,campo_m)]

```
INSERT INTO CLIENTES (NIF, NOMBRE, DOMICILIO, TLF, CIUDAD)
VALUES ('51639936K', 'MALDONADO GÓMEZ CARLOS', 'C/ LUNA, 48', '913683060',
'MADRID');
```

Añade una nueva fila en la tabla *CLIENTES*.

La sentencia SQL *SELECT*

La sentencia *SELECT* se utiliza para recuperar un conjunto de filas a partir de una o varias tablas. Tiene el siguiente formato:

SELECT campos_a_seleccionar FROM tabla [WHERE condición]

```
SELECT NOMBRE, TLF FROM CLIENTES  
WHERE CIUDAD='MADRID';
```

Devuelve una relación con el nombre y teléfono de los clientes de Madrid.

NOMBRE	TLF	
LOPEZ VAL SOLEDAD	915-829-394	
DELGADO ROBLES MIGUEL	913-859-293	
GUTIERREZ PEREZ ROSA	919-592-932	

La sentencia SQL *UPDATE*

La sentencia *UPDATE* se utiliza para modificar uno o varios campos de uno o varios registros. Tiene el siguiente formato:

*UPDATE nombre_tabla SET [campo_1 = valor_1, ..., campo_m= valor_m]
[WHERE condición]*

```
UPDATE CLIENTES SET DOMICILIO='C/ SANTA ENGRACIA, 17 3ºA', TLF='913802060'  
WHERE NOMBRE='DELGADO PEREZ MARISA';
```

Modifica el domicilio y teléfono del cliente 'DELGADO PEREZ MARISA'.

La sentencia SQL *DELETE*

La sentencia *DELETE* se utiliza para eliminar una o varias filas o registros. Tiene el siguiente formato:

DELETE FROM nombre_tabla [WHERE condición]

```
DELETE FROM CLIENTE WHERE NOMBRE='DELGADO PEREZ MARISA';
```

Elimina la fila correspondiente al cliente 'DELGADO PEREZ MARISA'.

Las sentencias SQL deben finalizar con un punto y coma. No es obligatorio si solo vamos a ejecutar una sentencia, pero cuando vamos a ejecutar varias sentencias SQL en bloque, el punto y coma sirve como delimitador para marcar que termina una sentencia y comienza la siguiente.

Los sublenguajes de SQL

SQL además está dividido en dos sublenguajes:

- **DDL (Data Definition Language):** formado por el conjunto de sentencias SQL que tienen que ver con la creación y modificación de la estructura de la base de datos (crear tablas, borrar tablas, etc). Las sentencias que forman parte de este sublenguaje son: *CREATE*, *ALTER* y *DROP*.
- **DML (Data Management Language):** formado por el conjunto de sentencias SQL que tienen que ver con la manipulación de los datos (añadir registros, borrar registros, modificar registros, recuperar registros). Las sentencias que forman parte de este subconjunto son: *INSERT*, *SELECT*, *UPDATE* y *DELETE*.

Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- **Una base de datos es un “almacén”** que nos permite guardar grandes cantidades de información de forma organizada para que luego la podamos encontrar y utilizar fácilmente.
- Una base de datos está formada por una o más **tablas**.
- Cada tabla está compuesta por **filas (registros)** y **columnas (campos)**.
- Cada tabla suele tener una **clave principal**, que identifica a cada uno de los registros. La clave principal no podrá tener valores duplicados. La clave principal de una tabla sirve además para establecer interrelaciones con otras tablas de la base de datos.
- **SQL** es un lenguaje estándar para almacenar, manipular y recuperar información de una base de datos.
- Al conjunto de operaciones que podemos realizar con el lenguaje SQL se le denomina **CRUD (Create, Read, Update, Delete)**.

2.2. Instalación de MySQL y creación de la primera BD



Índice

Objetivos	4
Primeros pasos con MySQL	5
Introducción y conceptos básicos	5
Instalación y configuración de MySQL	5
El lenguaje de definición de datos (DDL)	7
Sublenguaje DDL	7
Tipos de datos numéricos	7
Tipos de datos para fechas y horas	8
Tipos de datos para guardar cadenas de texto	9
Creación, modificación y borrado de tablas	11
Abrir MySQL Workbench y crear el esquema de la base de datos	11
Crear tablas	14
Modificar y borrar tablas	16
ALTER TABLE – ADD COLUMN (añadir atributo)	16
ALTER TABLE – ALTER COLUMN (modificar atributo)	17
ALTER TABLE – DROP COLUMN (borrar atributo)	17
Creación, modificación y borrado de índices	18
¿Qué es un índice?	18
Crear índices	19
Modificar índices	20
Borrar índices	20
Restricciones	21
Restricciones de integridad	21
Los tipos de datos	21
Restricción UNIQUE	22
Restricción NOT NULL	22
Restricción PRIMARY KEY	22
Restricción FOREIGN KEY	23
Restricción AUTO_INCREMENT	24
Restricciones ENUM y SET	25

Ejemplo práctico	26
Diseño físico de la BD FERRETERIA	26
Despedida	30
Resumen.....	30

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Instalar MySQL y establecer la configuración básica.
- Crear el diseño físico de una base de datos, utilizando el lenguaje de definición de datos y el SGBD MySQL.
- Preparar los recursos necesarios para acometer la lección 3 de esta misma unidad.

Primeros pasos con MySQL

Introducción y conceptos básicos

MySQL es el SGBD relacional que más se utiliza actualmente, sobre todo en entornos de desarrollo web.

Los más importantes proyectos de gestores de contenidos, como WordPress, Joomla o Prestashop, utilizan bases de datos MySQL.

MySQL actúa como **servidor de bases de datos a partir de instancias**, también denominadas conexiones MySQL.

Cada instancia o conexión puede alojar varios **esquemas (schemas)** y atenderá conexiones de clientes que deseen acceder a dichos esquemas a través del nombre del servidor o dirección IP y un puerto de comunicaciones.

Un esquema es una **estructura que engloba a una base de datos y otros objetos asociados a ella**, como vistas, procedimientos almacenados, lanzadores (*triggers*), etc.

Instalación y configuración de MySQL

MySQL Workbench es una herramienta visual de diseño que está integrada en el SGBD MySQL, además de una interfaz de usuario para su gestión de forma más rápida y cómoda.

A continuación, aprenderás a descargar, instalar y configurar el servidor de MySQL y sus herramientas asociadas, incluyendo MySQL Workbench.

Existen varias versiones de MySQL. Estos dos vídeos te mostrarán cómo descargar, instalar y configurar la versión *Community*.

El siguiente vídeo muestra los pasos para descargar MySQL desde Internet:

<https://vimeo.com/telefonicaed/review/265581955/21bb7d9ae2>

Durante la instalación de MySQL pasarás por tres etapas: **instalación de requisitos, instalación de MySQL y configuración de MySQL**.

En la etapa de configuración te pedirá el *password* (contraseña) del usuario *root* (administrador). El usuario *root* se crea por defecto durante la instalación. **Utiliza un *password* del que luego te acuerdes.**

El siguiente vídeo muestra los pasos a seguir para instalar y configurar MySQL:

<https://vimeo.com/telefonicaed/review/265581979/cbcec1659b>

El lenguaje de definición de datos (DDL)

Sublenguaje DDL

Puesto que el lenguaje de definición de datos (DDL) nos permite, entre otras cosas, realizar el diseño físico de la base de datos por cada tabla o relación que creemos, habrá que determinar sus atributos o columnas.

Para cada atributo será necesario establecer su dominio, es decir, el tipo y rango de valores que puede almacenar.

Pero antes de aprender a crear tablas utilizando el lenguaje DDL es necesario conocer los **tipos de datos que maneja el lenguaje SQL**.

Tipos de datos numéricos

SQL cuenta con gran cantidad de tipos de datos para almacenar números enteros o reales.

TINYINT:

Ocupa 1 byte y su dominio es entre -128 y 127 o entre 0 y 255 (utilizando la restricción UNSIGNED).

SMALLINT:

Ocupa 2 bytes y su dominio es entre -32.768 y 32.767 o entre 0 y 65.535 (utilizando la restricción UNSIGNED).

MEDIUMINT:

Ocupa 3 bytes y su dominio es entre -8.388.608 y 8.388.607 o entre 0 y 16.777.215 (utilizando la restricción UNSIGNED).

INT (INTEGER):

Ocupa 4 bytes y su dominio es entre -2.147.483.648 y 2.147.483.647 o entre 0 y 4.294.967.295 (utilizando la restricción UNSIGNED).

BIGINT:

Ocupa 8 bytes y su dominio es entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807 o entre 0 a 18.446.744.073.709.551.615 (utilizando la restricción UNSIGNED).

FLOAT:

Ocupa 4 bytes y almacena un número pequeño en coma flotante de precisión simple. Dominio entre -3.402823466E+38 y -1.175494351E-38, 0, y desde 1.175494351E-38 a 3.402823466E+38 (utilizando la restricción UNSIGNED).

DOUBLE (REAL):

Ocupa 8 bytes. Almacena números de coma flotante con precisión doble. Dominio entre -1.7976931348623157E+308 y -2.2250738585072014E-308, 0, y desde 2.2250738585072014E-308 a 1.7976931348623157E+308 (utilizando la restricción UNSIGNED).

DECIMAL (M, D):

Ocupa M+2 bytes si D>0 y M+1 bytes si D=0. Números en coma flotante almacenados como cadena.

BIT (BOOL, BOOLEAN):

Número entero con valor 0 o 1.

Tipos de datos para fechas y horas

SQL dispone de varios tipos de datos para almacenar fechas, horas o instantes en el tiempo (fecha+hora).

DATE:

Ocupa 3 bytes. Fecha en formato año-mes-día, con rango desde '1000-01-01' hasta '9999-12-31'.

DATETIME:

Ocupa 8 bytes. Fecha en formato año-mes-día y hora en formato horas-minutos-segundos. Rango de '1000-01-01 00:00:00' y '9999-12-31 23:59:59'.

TIME:

Ocupa 3 bytes. Para almacenar una hora en formato 'hh:mm:ss'.

YEAR:

Ocupa 1 byte. Almacena un año con 2 o 4 dígitos de longitud. Por defecto son 4 dígitos, para 2 dígitos se expresa YEAR(2). Rango con 4 dígitos de 1901 a 2155. Rango con 2 dígitos 1970 a 2069 (70 a 69).

TIMESTAMP:

Ocupa 4 bytes. Fecha y hora en formato UCT. Rango entre '1970-01-01 00:00:01' y '2038-01-19 03:14:07'.

Tipos de datos para guardar cadenas de texto

SQL dispone de muchísimos tipos de datos para almacenar textos de tamaño fijo o variable.

CHAR(N):

Cadena de longitud fija (N caracteres). Límite 255 caracteres. Si se declara CHAR(25) y se almacena el valor 'PEPE', ocupará 25 caracteres aunque sólo se hayan utilizado 4.

VARCHAR(N):

Cadena de longitud variable (N caracteres). Límite 255 caracteres. Si se declara VARCHAR(25) y se almacena el valor 'PEPE', sólo ocupará 4 caracteres.

TINYTEXT:

Cadena con un máximo de 255 caracteres. No distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

TINYBLOB:

Cadena con un máximo de 255 caracteres. Distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

TEXT:

Cadena con un máximo de 65.535 caracteres. No distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

BLOB:

Cadena con un máximo de 65.535 caracteres. Distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

MEDIUMTEXT:

Cadena con un máximo de 16.777.215 caracteres. No distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

MEDIUMBLOB:

Cadena con un máximo de 16.777.215 caracteres. Distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

LONGTEXT:

Cadena con un máximo de 4.294.967.295 caracteres. No distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

LONGBLOB:

Cadena con un máximo de 4.294.967.295 caracteres. Distingue entre mayúsculas y minúsculas a la hora de realizar ordenaciones o búsquedas.

ENUM:

Campo que sólo puede tomar un valor de una lista específica. Acepta hasta 65.535 valores distintos.

SET:

Campo que sólo puede tomar un valor de una lista específica. Acepta hasta 64 valores distintos.

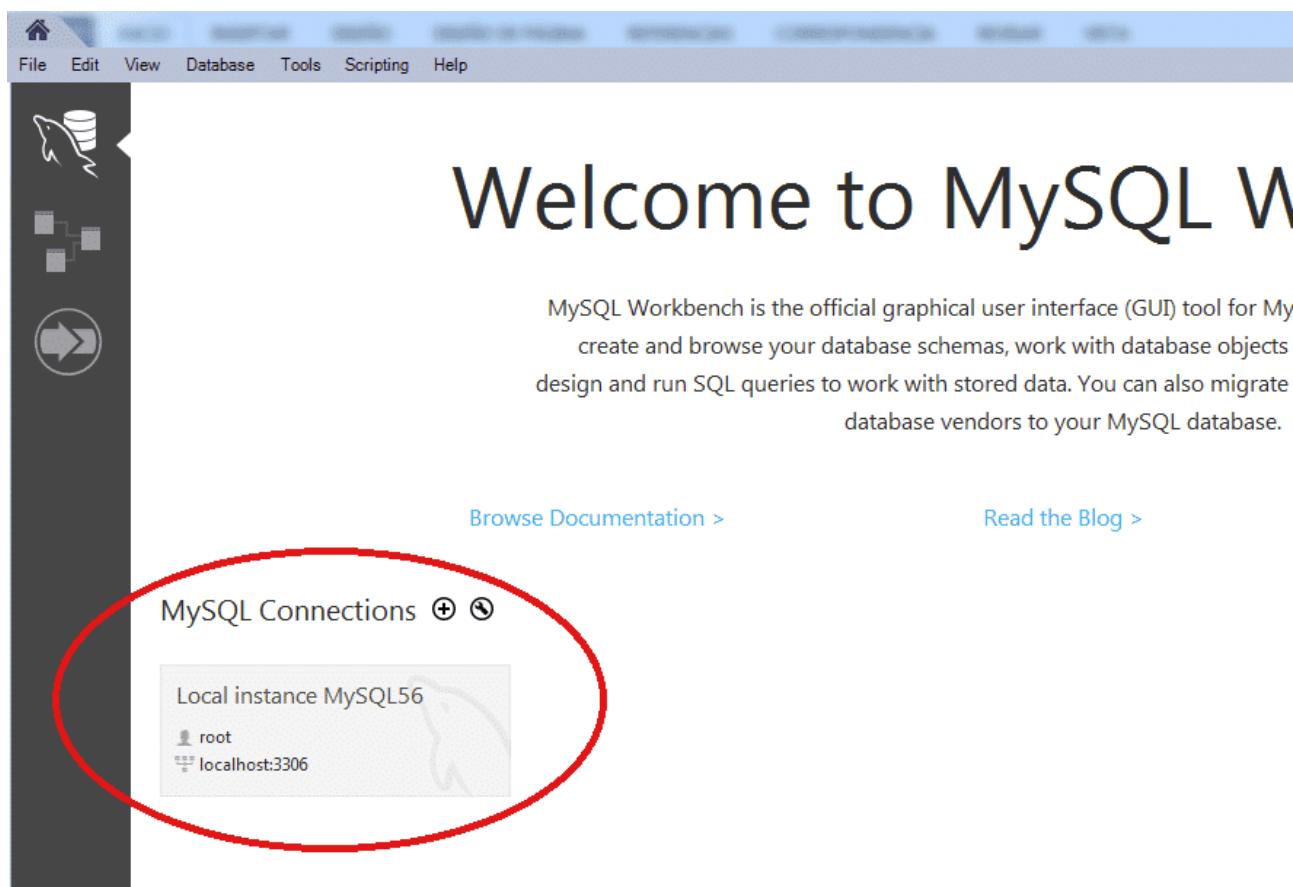
Creación, modificación y borrado de tablas

Abrir MySQL Workbench y crear el esquema de la base de datos

Antes de ponernos a crear tablas, es necesario comenzar por crear un esquema y ponerlo en uso.

Para comenzar abre MySQL Workbench.

La pantalla será similar a la mostrada en la imagen.

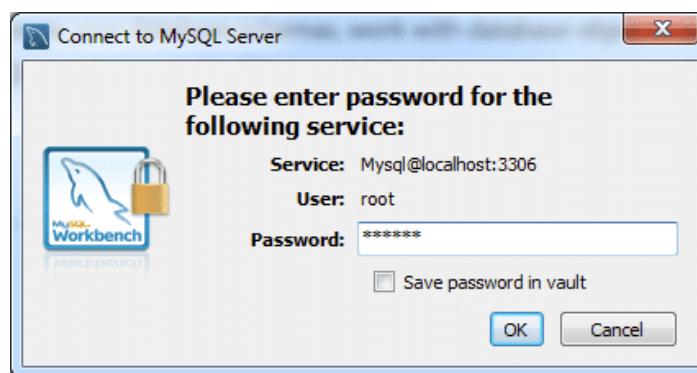


En la zona que se muestra en la imagen con un óvalo rojo, es donde aparece la instancia o *Connection* de MySQL. Podrían existir varias.

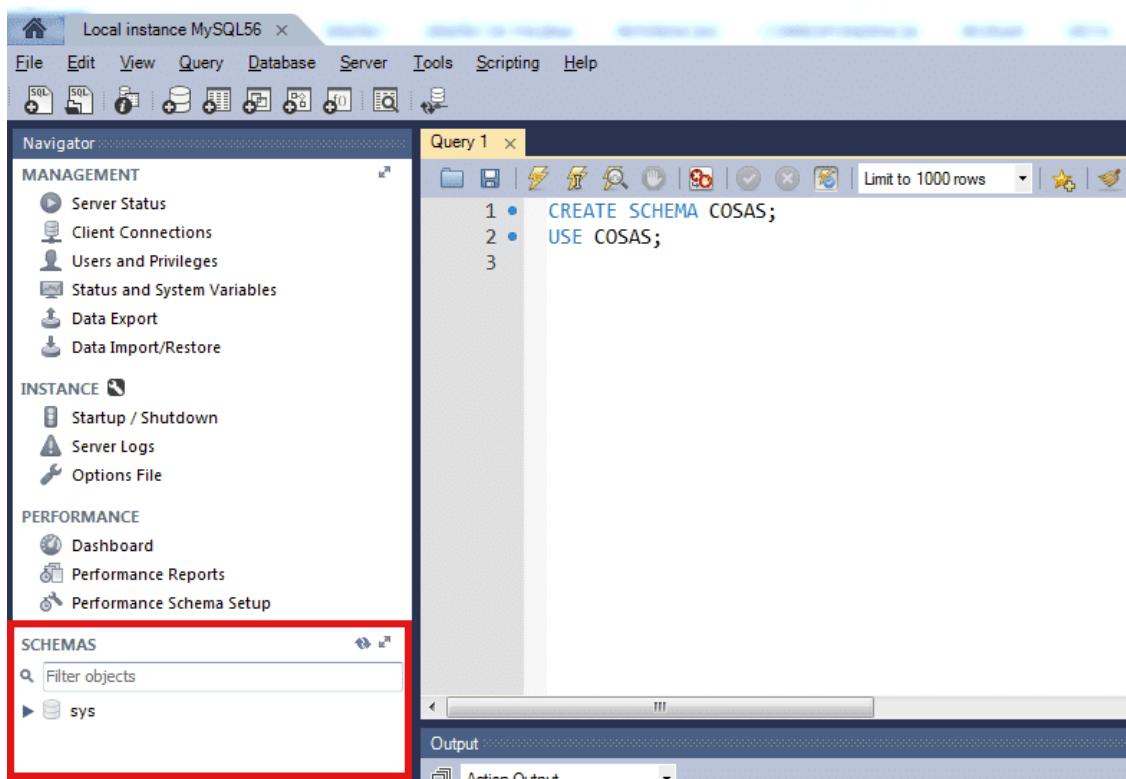
Observa que a la derecha del texto “MySQL Connections” aparece un ícono con un signo + que te permitirá crear una nueva instancia si lo deseas, y otro ícono con una llave inglesa que te permitirá modificar la configuración de la instancia si es necesario.

A continuación, aparecen el nombre de la instancia, el usuario autorizado, el servidor y el puerto. Para comenzar a trabajar **debes hacer clic sobre el nombre de la instancia. En la imagen aparece como “Local instance MySQL56”.**

A continuación, **te pedirá la contraseña para el usuario root** que especificaste durante la instalación.



Pulsa OK y llegarás al entorno de trabajo de MySQL Workbench, donde podrás comenzar en breve a escribir tus consultas SQL.



El panel de la izquierda te brinda las distintas herramientas disponibles en MySQL Workbench.

Resaltado en rojo está el área donde se van mostrando los distintos esquemas. Por el momento sólo se ve el esquema *sys* (base de datos del sistema).

El área de la derecha es el editor para escribir SQL; para comenzar puedes escribir lo siguiente:

```
CREATE SCHEMA COSAS;  
USE COSAS;
```

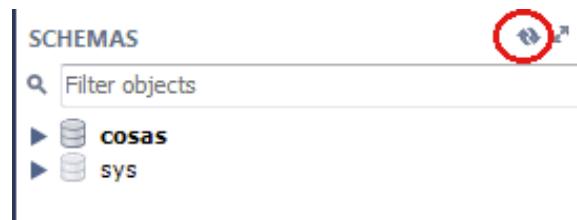
El carácter de **punto y coma representa el final de línea** o sentencia. Es necesario cuando ejecutamos varias sentencias para identificar cuando termina una y comienza la otra.

Luego, **ejecuta** ambas sentencias **haciendo clic en el icono** que está representado por un **rayo amarillo**.

Por el momento la idea no es que crees una base de datos normalizada completa, sino que comiences a familiarizarte con el entorno de MySQL Workbench y los tipos de datos. Por esa razón, hemos llamado al esquema *Cosas*, ya que será un almacén de tablas de prueba y aprendizaje.

Al ejecutar estas sentencias, primero has creado un esquema llamado *Cosas* y luego lo has puesto en uso, lo que significa que cuando comiences a crear tablas e índices se crearán en este esquema y no en otro.

El área de esquemas te mostrará el nuevo esquema una vez que hagas clic en el botón “Actualizar”, que está resaltado con un óvalo rojo en la siguiente imagen:



El esquema que se muestra en negrita es el esquema en uso. Debes tener esto siempre en cuenta para no crear por error objetos en el esquema incorrecto.

Crear tablas

Para crear nuevas relaciones o tablas se utiliza la sentencia ***CREATE TABLE***.

Esta sentencia tiene la siguiente **estructura**:

```
CREATE TABLE nombre_tabla (atributo1 tipo, atributo2 tipo,... atributoN tipo)
```

Como ejemplo, vamos a crear una tabla llamada *Amigos*, que permitirá almacenar el nombre, dirección y teléfono de cada uno de nuestros amigos e identificar mediante una etiqueta dónde lo conocemos (en la escuela, en el trabajo, en el gimnasio o en otro lugar).

```
CREATE TABLE AMIGOS (
    Nombre CHAR(25),
    Direccion VARCHAR(100),
    Tlf VARCHAR(15),
    Etiqueta SET('Escuela','Trabajo','Gimnasio','Otros')
);
```

Para comprobar que las restricciones de dominio asignadas en la tabla *Amigos* están funcionando, puedes insertar una nueva fila con una sentencia **INSERT** como esta:

```
INSERT INTO AMIGOS VALUES ('Carlos', 'C/ Oca, 25', '616616616', 'Barrio');
```

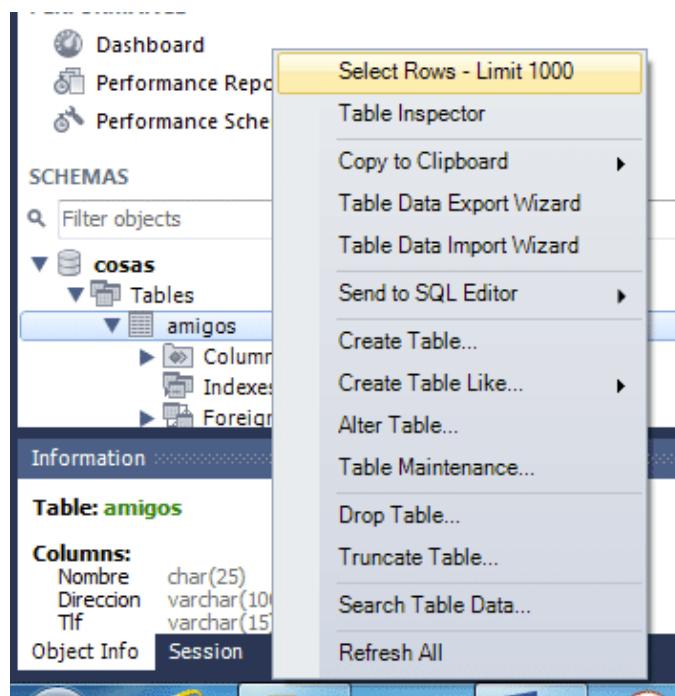
Comprobarás que da error y no añade la nueva fila porque el valor 'Barrio' está fuera del conjunto de valores del dominio ('Escuela', 'Trabajo', 'Gimnasio', 'Otros').

Prueba ahora a cambiar el valor por uno de los que forman parte del dominio, como en el siguiente ejemplo:

```
INSERT INTO AMIGOS VALUES ('Carlos', 'C/ Oca, 25', '616616616', 'Trabajo');
```

Ahora sí que ha funcionado.

Para comprobar que el nuevo registro ha sido insertado correctamente puedes hacer doble clic sobre el nombre del esquema *Cosas*, luego en *Tables* y luego en *Amigos*. Si no aparece, tendrás que volver a usar el icono actualizar.



Si haces un clic derecho sobre la tabla *Amigos* aparece un menú contextual como el que ves en la imagen.

La opción “Select Rows ...” te mostrará los registros o filas.

ACTIVIDADES

A continuación, te proponemos dos ejercicios en los que tendrás que crear una tabla. Para cada tabla deberás consultar la lista de tipos de datos para escoger el tipo idóneo en cada caso.

Después de crear la tabla, inserta un registro como muestra.

1. Tabla *Alumno* con los siguientes campos: *Nombre* (cadena de longitud variable con un ancho máximo de 25), *Edad* (valor numérico entero entre 0 y 255), *Tlf* (cadena de longitud fija con un ancho de 12), *Mensualidad* (número real de 4 bytes que no admite negativos), *NumeroMatricula* (número entero positivo con valor máximo 65535) y *fechaMatricula* (fecha de 3 bytes de ocupación con máximo valor '9999-12-31').

Solución:

```
CREATE TABLE ALUMNO (
    Nombre VARCHAR(25),
    Edad TINYINT UNSIGNED,
    Tlf CHAR(12),
    Mensualidad FLOAT UNSIGNED,
    FechaMatricula DATE
);
INSERT INTO ALUMNO VALUES (
```

```
'Perico de los Palotes',
25,
'913670488',
53.50,
3524,
'2017-12-11'
);
```

2. Tabla *Candidato* con los siguientes campos: *Nombre* (cadena de longitud variable con un ancho máximo de 25), *FechaNacimiento* (fecha de 3 bytes de ocupación con máximo valor '9999-12-31'), *Tlf* (cadena de longitud fija con un ancho de 12), *FechaHoraPresentacion* (fecha y hora de 4 bytes) y *Curriculum* (cadena máximo 16.777.215 caracteres que no distinga entre mayúsculas y minúsculas).

Solución:

```
CREATE TABLE Candidato (
    Nombre VARCHAR(25),
    FechaNacimiento DATE,
    Tlf CHAR(12),
    FechaHoraPresentacion TIMESTAMP,
    Curriculum MEDIUMTEXT
);
INSERT INTO Candidato VALUES (
    'Pedro Delta',
    '1981-12-03',
    '913676767',
    '2017-12-15',
    'Aquí el curriculum de Pedro Delta'
);
```

Modificar y borrar tablas

Para eliminar o modificar la estructura de una tabla debes utilizar la sentencia ALTER.

ALTER TABLE – ADD COLUMN (añadir atributo)

Añadir nueva columna en una tabla existente.

Formato

```
ALTER TABLE nombreTabla
ADD COLUMN atributo tipo
```

Ejemplo

```
ALTER TABLE Candidato ADD COLUMN Ciudad VARCHAR(50);
```

Has añadido a la tabla *Candidato* el atributo o columna *Ciudad*. Para las filas que ya estuvieran guardadas en la tabla establecerá la *Ciudad* con el valor NULL. El nuevo atributo quedará en la última columna.

Si queremos situar el nuevo atributo en una columna concreta, podemos utilizar las palabras reservadas BEFORE o FIRST delante del nombre de una columna.

```
ALTER TABLE ALUMNO ADD COLUMN Ciudad VARCHAR(50)  
AFTER Nombre;
```

ALTER TABLE – ALTER COLUMN (modificar atributo)

Modificar la definición de una columna existente.

En el siguiente ejemplo, modificamos la definición del atributo *Ciudad* dejándolo en 40 caracteres.

```
ALTER TABLE ALUMNO MODIFY COLUMN Ciudad VARCHAR(40);
```

ALTER TABLE – DROP COLUMN (borrar atributo)

Eliminar una columna.

En el siguiente ejemplo, eliminamos el campo *Ciudad* de la tabla *Alumno*.

```
ALTER TABLE ALUMNO DROP COLUMN Ciudad;
```

Creación, modificación y borrado de índices

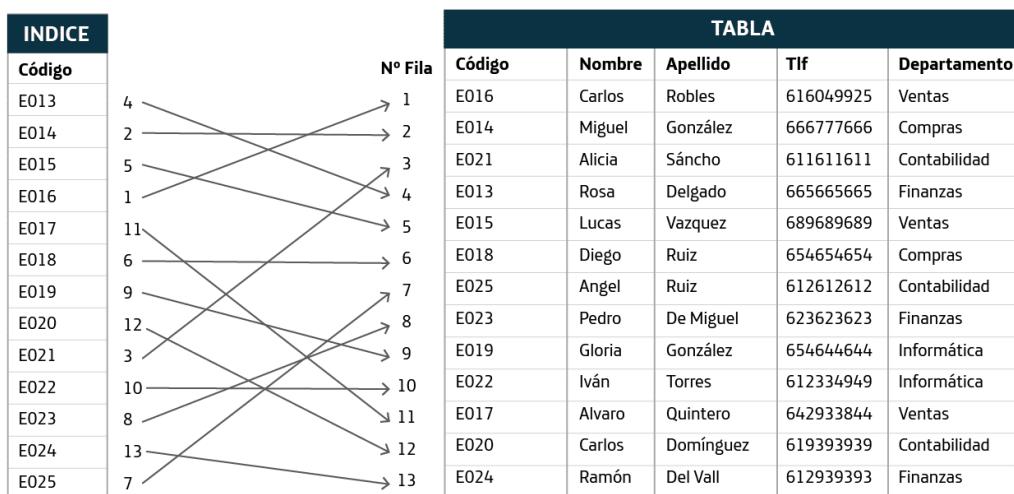
¿Qué es un índice?

Los índices de las tablas son estructuras que permiten agilizar las consultas.

Para que comprendas mejor su funcionalidad puedes compararlo con el índice de cualquier libro.

Se tarda más en buscar un contenido pasando página tras página, que consultando primero en el índice y luego accediendo directamente a la página.

Cada índice es como una nueva tabla formada por una clave, que estará formada por un solo atributo o la unión de varios atributos de la tabla original.



Funcionamiento de los índices.

Cuando realizamos la consulta de un empleado a partir del código, el SGBD buscará en el índice y accederá directamente a la posición de la tabla a la que apunta. Esta operación es transparente para el usuario. Nosotros realizamos la consulta de la misma manera, tanto si hay índice como si no lo hay.

Crear índices

Para crear un índice utilizamos la sentencia **CREATE INDEX**.

Esta sentencia tiene la siguiente estructura:

```
CREATE INDEX nombre_indexe  
ON nombre_tabla (column1, column2, ...);
```

Ejemplo

Creamos una tabla llamada *Articulo* y después un índice para la tabla *Articulo* cuya clave es el atributo *Codigo*.

```
CREATE TABLE Articulo (  
    Codigo CHAR(4),  
    Descripcion VARCHAR(100),  
    Precio FLOAT,  
    Stock INT  
);  
  
CREATE INDEX idx_Codigo  
ON Articulo(Codigo);
```

La creación del índice *idx_codigo* agiliza las búsquedas de artículos a partir del código, pero no evita que puedan duplicarse códigos. Para crear un índice que, además, añada la restricción UNIQUE, la sentencia SQL sería así:

```
CREATE UNIQUE INDEX idx_Codigo  
ON Articulo(Codigo);
```

CREATE INDEX no se utiliza para crear la clave primaria. Para eso se emplea la cláusula PRIMARY KEY en el momento en que se crea o modifica la tabla.

A continuación, veremos tres alternativas para crear la tabla *Articulo* y establecer el atributo *Codigo* como clave primaria, aunque el resultado será exactamente igual en los tres casos.

1. Como una restricción aplicada al atributo *Codigo*

```
CREATE TABLE Articulo (  
    Codigo CHAR(4) PRIMARY KEY,  
    Descripcion VARCHAR(100),  
    Precio FLOAT,  
    Stock INT  
);
```

2. Como una cláusula que recibe como argumento al atributo de clave primaria

```
CREATE TABLE Articulo (
    Codigo CHAR(4),
    Descripcion VARCHAR(100),
    Precio FLOAT,
    Stock INT,
    PRIMARY KEY(Codigo)
);
```

3. A posteriori con la sentencia ALTER

```
CREATE TABLE Articulo (
    Codigo CHAR(4),
    Descripcion VARCHAR(100),
    Precio FLOAT,
    Stock INT
);

ALTER TABLE Articulo
MODIFY COLUMN Codigo CHAR(4) PRIMARY KEY;
```

Modificar índices

Ahora aprenderemos que es posible cambiar el nombre de un índice.

Para ello, aplicaremos el siguiente formato de la sentencia ALTER.

```
ALTER TABLE nombre_tabla
RENAME INDEX nombre_indice TO nombre_indice_nuevo;
```

Ejemplo

```
ALTER TABLE Articulo
RENAME INDEX idx_Codigo TO index_Codigo;
```

Borrar índices

Para eliminar un índice se utiliza la sentencia ALTER con la cláusula DROP aplicando el siguiente formato:

```
ALTER TABLE nombre_tabla
DROP INDEX nombre_indice
```

Ejemplo

```
ALTER TABLE Articulo
DROP INDEX idx_Codigo;
```

Si lo que deseamos es borrar la clave primaria:

```
ALTER TABLE Articulo
DROP PRIMARY KEY;
```

Restricciones

Restricciones de integridad

Las restricciones de integridad de una base de datos definen cómo se encargará el DBMS de exigir ciertas reglas.

Estas reglas establecen los **valores permitidos en determinados atributos o campos** para exigir la integridad y coherencia de la base de datos en su conjunto.

Los tipos de datos

El mero hecho de crear una tabla y asignar un tipo de dato a cada atributo lleva consigo una serie de restricciones de integridad, inherentes al modelo relacional.

Observa de nuevo la sentencia para la creación de la tabla *Articulo*:

```
CREATE TABLE Articulo (
    Código CHAR(4),
    Descripción VARCHAR(100),
    Precio FLOAT,
    Stock INT
);
```

El motor de MySQL garantizará que se cumplan las siguientes restricciones:

- No se podrán introducir más de 4 caracteres en el código de producto.
- No se podrán introducir más de 100 caracteres en la descripción del producto.
- El precio deberá ser un valor numérico, no admitirá letras.
- El stock deberá ser un número entero.

Todas estas son **restricciones de dominio**, ya que determinan el conjunto de valores válidos para un atributo.

Restricción UNIQUE

La restricción UNIQUE no permite que el atributo al que se aplica tenga valores repetidos.

```
CREATE TABLE AMIGOS (
    Nombre CHAR(25) UNIQUE,
    Direccion VARCHAR(100),
    Tlf VARCHAR(15),
    Etiqueta SET('Escuela','Trabajo','Gimnasio','Otros')
);
```

¡Ojo! No se permitirá introducir dos filas en la tabla *Amigos* con el mismo valor del atributo *Nombre*.

Restricción NOT NULL

La restricción NOT NULL no permite que el atributo al que se aplica se deje vacío.

```
CREATE TABLE AMIGOS (
    Nombre CHAR(25) NOT NULL,
    Direccion VARCHAR(100),
    Tlf VARCHAR(15),
    Etiqueta SET('Escuela','Trabajo','Gimnasio','Otros')
);
```

No se admiten valores nulos en el atributo *Nombre*. Eso significa que tendremos que asignar un valor obligatoriamente.

Hay que tener en cuenta que un campo de texto sin llenar tendrá por defecto valor NULL.

Restricción PRIMARY KEY

El hecho de definir un atributo como PRIMARY KEY (clave primaria) lleva consigo una importante restricción.

```
CREATE TABLE Articulo (
    Codigo CHAR(4) PRIMARY KEY,
    Descripcion VARCHAR(100),
    Precio FLOAT,
    Stock INT
);
```

Establecer el campo *Código* como PRIMARY KEY garantiza:

- **Que no habrá dos productos con el mismo código**, ya que lleva implícita la restricción UNIQUE.
- **Que no se podrá llenar una fila dejando el código de producto vacío**, ya que lleva implícita la restricción NOT NULL (no admite valores nulos).

Restricción FOREIGN KEY

La restricción FOREIGN KEY permite garantizar la integridad referencial cuando existe una asociación entre dos relaciones o tablas.

Veamos un ejemplo.

Aquí tenemos una tabla *Articulo* que contiene los artículos de un almacén, y otra relación o tabla *VentaContado* que registra las veces que se vende uno de los artículos del almacén.

```
CREATE TABLE Articulo (
    Código CHAR(4) PRIMARY KEY,
    Descripción VARCHAR(100),
    Precio FLOAT,
    Stock INT
);

CREATE TABLE VentaContado (
    NúmeroVenta INT AUTO_INCREMENT PRIMARY KEY,
    Código_Articulo CHAR(4),
    Unidades INT,
    Precio FLOAT,
    FOREIGN KEY (Código_Articulo) REFERENCES Articulo(Código)
)
```

En el anterior ejemplo, *CódigoArticulo* es la clave extranjera o ajena que establece la asociación con la tabla *Articulo*.

Sin embargo, en el ejemplo siguiente, la restricción FOREIGN KEY está garantizando que no se añada ninguna *VentaContado* con un código de artículo que no esté previamente registrado en la tabla *Articulo*.

Otra alternativa es crear la tabla *VentaContado* sin establecer la clave ajena y hacerlo después con ayuda de la sentencia ALTER TABLE de la siguiente manera:

```
CREATE TABLE VentaContado (
    NúmeroVenta INT AUTO_INCREMENT PRIMARY KEY,
    Código_Articulo CHAR(4),
    Unidades INT,
    Precio FLOAT
);
```

```
ALTER TABLE VentaContado  
ADD FOREIGN KEY (Codigo_Articulo) REFERENCES Articulo(Codigo);
```

IMPORTANTE: Cada clave ajena es un índice que tendrá un nombre que lo identifique. Sin embargo, en ninguno de los ejemplos anteriores hemos decidido qué nombre deseamos que tenga dicho índice. El motor del SGBD de MySQL ha elegido el nombre por nosotros.

En la sentencia ALTER TABLE es posible indicar el nombre que deseamos para el índice, de la siguiente manera:

```
ALTER TABLE VentaContado  
ADD CONSTRAINT FK_ArticuloVenta  
FOREIGN KEY (Codigo_Articulo) REFERENCES Articulo(Codigo);
```

FK_ArticuloVenta es ahora el nombre del índice de clave ajena. Conocer el nombre nos permitirá poder eliminar el índice en caso necesario, de la siguiente manera:

```
ALTER TABLE VentaContado  
DROP FOREIGN KEY FK_ArticuloVenta;
```

Restricción AUTO_INCREMENT

La cláusula AUTO_INCREMENT permite que un atributo de tipo numérico entero adquiera un valor automático, que va incrementándose (contador).

Por defecto, el primer valor será 1 e irá incrementándose, sumando 1 en cada registro.

```
CREATE TABLE Empleado (  
    ID int AUTO_INCREMENT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    Edad int,  
    CHECK (Edad>=18)  
) ;
```

MySQL obliga a que sólo exista un atributo AUTO_INCREMENT en cada tabla y, además, debe ser clave.

Si queremos que el valor inicial sea distinto de 1 podemos utilizar la sentencia ALTER de la siguiente manera:

```
ALTER TABLE Empleado AUTO_INCREMENT=100;
```

AUTO_INCREMENT se considera una restricción porque de manera implícita **hace que el atributo cumpla con las restricciones NOT NULL y UNIQUE.**

Restricciones ENUM y SET

Las restricciones ENUM y SET permiten limitar la entrada de datos en un atributo a un conjunto de valores previamente establecido.

```
CREATE TABLE AMIGOS (
    Nombre CHAR(25),
    Direccion VARCHAR(100),
    Tlf VARCHAR(15),
    Etiqueta SET('Escuela','Trabajo','Gimnasio','Otros')
);
```

El campo *Etiqueta* sólo podrá contener los valores 'Escuela', 'Trabajo', 'Gimnasio' u 'Otros'. ENUM y SET se utilizan exactamente igual, pero presentan las siguientes diferencias:

ENUM

Campo que sólo puede tomar un valor de una lista específica. Acepta hasta 65.535 valores distintos.

SET:

Campo que sólo puede tomar un valor de una lista específica. Acepta hasta 64 valores distintos.

Ejemplo práctico

Diseño físico de la BD FERRETERIA

En este apartado vamos a crear la estructura de la base de datos FERRETERIA, que nos servirá como base para la parte práctica de la siguiente lección.

Y la crearemos conforme al siguiente diagrama de Modelo-Entidad-Relación.

Un Modelo-Entidad-Relación refleja las tablas (también llamadas entidades) que intervienen en una base de datos y las asociaciones existentes entre dichas tablas.

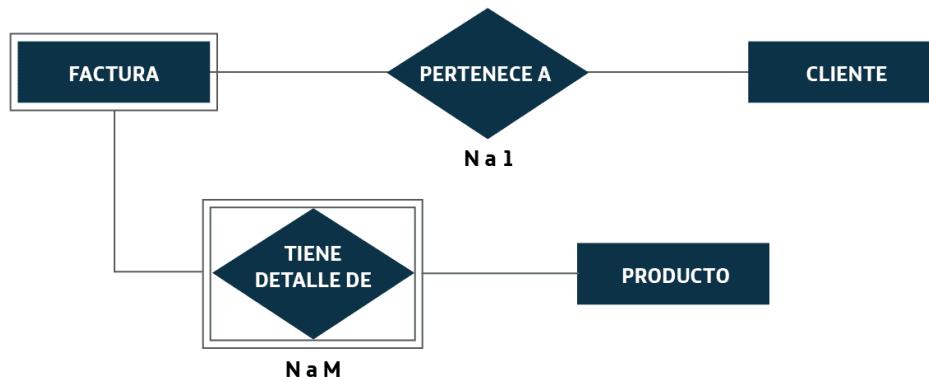


Diagrama de un Modelo Entidad-Relación.

Por el diagrama sabemos que existe una tabla llamada CLIENTE y una tabla llamada FACTURA.

También sabemos que cada factura pertenece a un cliente, es decir, existe una asociación entre ambas tablas. El N a 1 significa que un cliente puede tener N facturas, pero cada factura pertenece a un solo cliente.

También podemos comprobar en el diagrama que existe una tabla PRODUCTO que representa el almacén y que se relaciona con la factura, porque "una factura tiene detalle de producto".

Esta asociación es de N a M, porque cada factura incluye varios detalles de productos y cada producto está incluido en varias facturas. Esta asociación quedará finalmente reflejada físicamente en una tabla llamada DETALLE.

Ahora podemos realizar el diseño físico de la base de datos FERRETERIA con ayuda del lenguaje SQL.

1. Ejecuta este código en MySql para crear la estructura de la BD FERRETERIA.

```
CREATE SCHEMA FERRETERIA;
USE FERRETERIA;

CREATE TABLE CLIENTE (
    NIF CHAR(9) PRIMARY KEY,
    NOMBRE VARCHAR(25) NOT NULL,
    DOMICILIO VARCHAR(100),
    TLF VARCHAR(25),
    CIUDAD VARCHAR(50)
);

CREATE TABLE PRODUCTO (
    CODIGO CHAR(4) PRIMARY KEY,
    DESCRIPCION VARCHAR(100) NOT NULL,
    PRECIO FLOAT,
    STOCK FLOAT,
    MINIMO FLOAT,
    CHECK (PRECIO > 0)
);

CREATE TABLE FACTURA (
    NUMERO INT PRIMARY KEY,
    FECHA DATE,
    PAGADO BOOL,
    NIF CHAR(9),
    FOREIGN KEY (NIF) REFERENCES CLIENTE(NIF)
);

CREATE TABLE DETALLE (
    NUMERO INT,
    CODIGO CHAR(4),
    UNIDADES INT,
    PRECIO FLOAT,
    FOREIGN KEY (NUMERO) REFERENCES FACTURA(NUMERO),
    FOREIGN KEY (CODIGO) REFERENCES PRODUCTO(CODIGO),
    PRIMARY KEY (NUMERO, CODIGO)
);
```

2. Ejecuta este código en MySql para añadir filas o registros a la tabla CLIENTE.

```
INSERT INTO CLIENTE VALUES
('43434343A', 'DELGADO PEREZ MARISA', 'C/ MIRAMAR, 84 3ºA', '925-200-967',
'TOLEDO');

INSERT INTO CLIENTE VALUES
('51592939K', 'LOPEZ VAL SOLEDAD', 'C/ PEZ, 54 4ºC', '915-829-394', 'MADRID');

INSERT INTO CLIENTE VALUES
('51639989K', 'DELGADO ROBLES MIGUEL', 'C/ OCA, 54 5ºC', '913-859-293',
'MADRID');

INSERT INTO CLIENTE VALUES
```

```
('51664372R', 'GUTIERREZ PEREZ ROSA', 'C/ CASTILLA, 4 4ºA', '919-592-932',  
'MADRID');
```

3. Ejecuta esté código en MySql para añadir filas o registros a la tabla PRODUCTO.

```
INSERT INTO PRODUCTO VALUES  
('CAJ1', 'CAJA DE HERRAMIENTAS DE PLASTICO', 8.50, 4.00, 3);  
  
INSERT INTO PRODUCTO VALUES  
('CAJ2', 'CAJA DE HERRAMIENTAS DE METAL', 12.30, 3.00, 2);  
  
INSERT INTO PRODUCTO VALUES  
('MAR1', 'MARTILLO PEQUEÑO', 3.50, 5, 10);  
  
INSERT INTO PRODUCTO VALUES  
('MAR2', 'MARTILLO GRANDE', 6.50, 12, 10);  
  
INSERT INTO PRODUCTO VALUES  
('TOR7', 'CAJA DE 100 TORNILLOS DEL 7', 0.80, 20, 100);  
  
INSERT INTO PRODUCTO VALUES  
('TOR9', 'CAJA DE 100 TORNILLOS DEL 9', 0.80, 25, 100);  
  
INSERT INTO PRODUCTO VALUES  
('TUE7', 'CAJA DE 100 TUERCAS DEL 7', 0.50, 40, 100);  
  
INSERT INTO PRODUCTO VALUES  
('TUE9', 'CAJA DE 100 TUERCAS DEL 9', 0.50, 54, 100);
```

4. Ejecuta esté código en MySql para añadir filas o registros a la tabla FACTURA.

Observa cómo los valores introducidos para el campo NIF tienen correspondencia con los valores introducidos en la tabla CLIENTE. De esta manera, se establece la asociación de cada factura con el cliente al que pertenece.

```
INSERT INTO FACTURA VALUES  
(5440, '2017-09-05', true, '43434343A');  
  
INSERT INTO FACTURA VALUES  
(5441, '2017-09-05', true, '51639989K');  
  
INSERT INTO FACTURA VALUES  
(5442, '2017-09-06', false, '43434343A');  
  
INSERT INTO FACTURA VALUES  
(5443, '2017-10-10', true, '51639989K');  
  
INSERT INTO FACTURA VALUES  
(5444, '2017-10-13', true, '51664372R');  
  
INSERT INTO FACTURA VALUES  
(5445, '2017-10-14', false, '43434343A');
```

5. Ejecuta este código en MySql para añadir filas o registros a la tabla DETALLE.

Observa que los valores introducidos para los campos NUMERO y CODIGO tienen correspondencia con los valores introducidos en las tablas FACTURA y PRODUCTO. De esta manera, se establece la asociación de cada línea de detalle con la factura y el artículo al que pertenece.

```
INSERT INTO DETALLE VALUES (5440, 'CAJ2', 2, 12.3);
INSERT INTO DETALLE VALUES (5440, 'MAR1', 1, 3.50);
INSERT INTO DETALLE VALUES (5440, 'TOR7', 2, 0.80);
INSERT INTO DETALLE VALUES (5440, 'TUE7', 2, 0.50);
INSERT INTO DETALLE VALUES (5441, 'CAJ1', 1, 8.50);
INSERT INTO DETALLE VALUES (5442, 'CAJ1', 1, 8.50);
INSERT INTO DETALLE VALUES (5442, 'MAR1', 2, 3.50);
INSERT INTO DETALLE VALUES (5443, 'TOR7', 1, 0.80);
INSERT INTO DETALLE VALUES (5443, 'TUE7', 1, 0.50);
INSERT INTO DETALLE VALUES (5444, 'MAR2', 1, 12.0);
INSERT INTO DETALLE VALUES (5445, 'TOR7', 5, 0.80);
INSERT INTO DETALLE VALUES (5445, 'TOR9', 5, 0.80);
INSERT INTO DETALLE VALUES (5445, 'TUE7', 5, 0.50);
INSERT INTO DETALLE VALUES (5445, 'TUE9', 5, 0.50);
```

Ya tienes el terreno preparado para acometer con éxito la siguiente lección, donde podrás manipular la base de datos FERRETERIA desde un programa Java.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Antes de crear la primera base de datos necesitamos instalar un **DBMS**.
- **MySQL Workbench** es una **herramienta visual de diseño integrada en el SGBD MySQL**, además de una interfaz de usuario para su gestión de forma más rápida y cómoda.
- El **lenguaje de definición de datos (DDL)** forma parte del estándar SQL y consta de tres sentencias que permiten la creación, eliminación y modificación de tablas, índices y otros objetos de la base de datos.
- Sentencias DDL: **CREATE, ALTER y DROP**.
- El **lenguaje de definición de datos** es vital para la creación del diseño físico de una base de datos, y requiere del conocimiento de los distintos tipos de datos a la hora de definir los atributos de las distintas tablas.
- Existen muchos **tipos de datos en el estándar SQL** y se clasifican en tres categorías:
 - Tipos de datos **numéricos**.
 - Tipos de datos **para almacenar fechas y horas**.
 - Tipos de datos **para almacenar texto**.

2.3. Acceso a bases de datos con JDBC



Índice

Objetivos	3
Acceso a bases de datos desde java	4
El API JDBC	4
Primer proyecto con acceso a base de datos	7
Descargar el conector y crear el proyecto	7
Descargar el <i>driver</i> para MySQL.....	8
Importar el <i>driver</i> en el proyecto Java.....	9
Conectar con la base de datos.....	11
La cadena de conexión	12
Obtener un listado de clientes	14
Añadir un nuevo cliente	17
Usar PreparedStatement	19
Borrar un cliente	21
Programar la gestión de clientes.....	23
Despedida	28
Resumen.....	28

Objetivos

En esta unidad perseguimos los siguientes objetivos:

- Conocer las librerías de clases Java relacionadas con el acceso a bases de datos.
- Establecer conexión con una base de datos utilizando el API JDBC.
- Conocer las clases principales de la librería *java.sql*.

Acceso a bases de datos desde java

El API JDBC

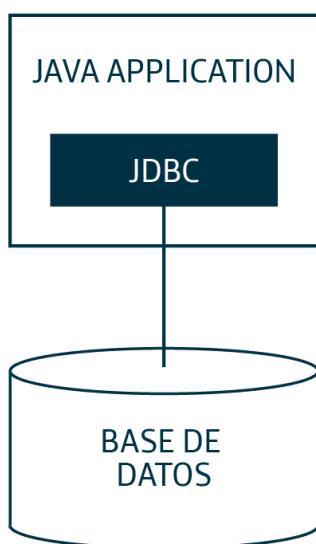
El API JDBC provee un mecanismo sencillo para acceder a bases de datos relacionales desde aplicaciones Java.

Se encuentra disponible en el paquete `java.sql`, que pertenece a la “JRE System Library”, por lo que no es necesario cargar ninguna librería externa.

Sin embargo, sí requiere incluir las sentencias ***import*** necesarias para importar las clases que vayamos a utilizar.

La principal característica de JDBC es que **permite independizar nuestro código Java de las peculiaridades de cada DBMS** (Data Base Management System), de modo que nuestro programa Java pueda funcionar exactamente igual accediendo a una base de datos MySQL, una base de datos SQL Server, Oracle, etc.

JDBC se sitúa como una capa dentro de nuestra aplicación Java, capaz de comunicarse con el DBMS.

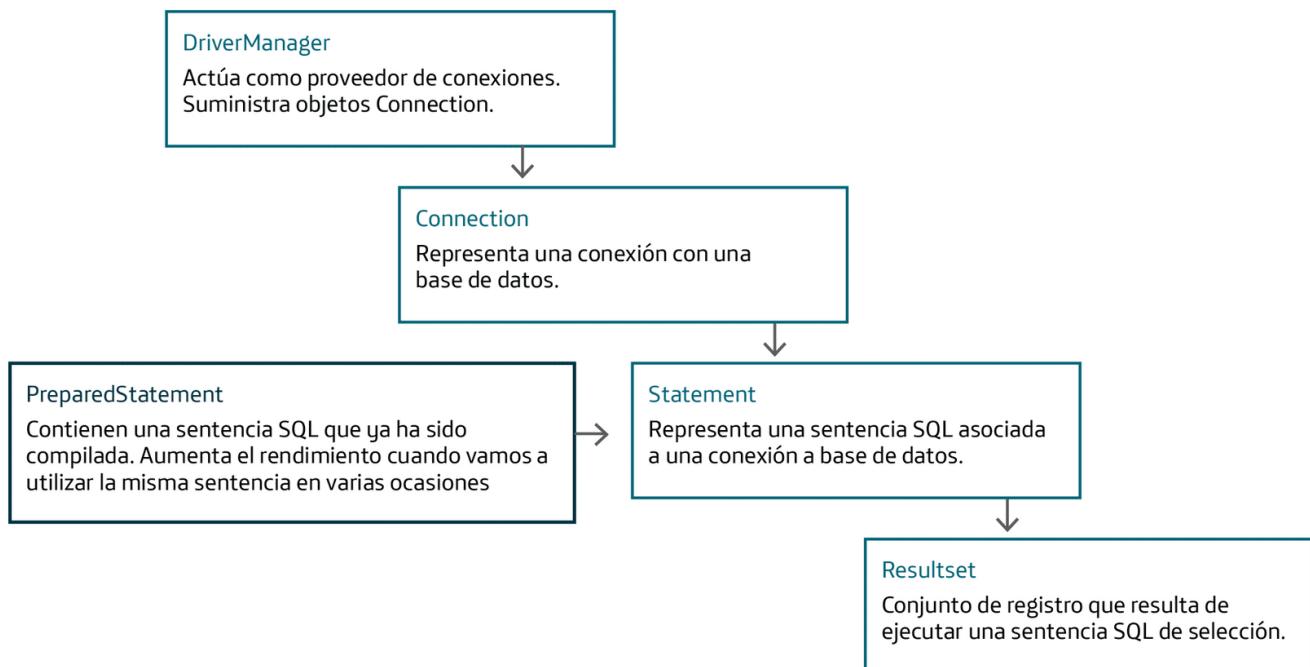


En función del DBMS que soporte la base de datos, JDBC necesitará un **driver o conector** especial para Java que habrá que añadir al proyecto.

Para los ejemplos de esta lección nos basaremos en una base de datos MySQL.

Desde nuestro programa Java realizamos llamadas estándar a métodos de las clases del API JDBC, y es el *driver* o conector el que se encarga de traducir estas llamadas a un lenguaje que entienda el DBMS.

Clases más importantes del API JDBC



Esquema de clases del API JDBC.

- **DriverManager**: como hemos comentado anteriormente, dentro del proyecto es necesario incluir un *driver* o conector por cada DBMS con el que nos queramos comunicar. Pues bien, será la clase *DriverManager* la que se encargue de gestionar estos *drivers* y de proveer al programa de las bases de datos que necesite.
- **Connection**: representa una conexión con una base de datos concreta, que será previamente suministrada por el *DriverManager*. La conexión con la base de datos se realiza a través de una cadena de conexión, que contendrá los datos de conexión necesarios.
- **Statement**: permite ejecutar una sentencia SQL, que podrá ser de selección de datos (*SELECT*), de actualización (*INSERT*, *DELETE*, *UPDATE*) o de definición (*CREATE*, *ALTER*, *DROP*).
- **PreparedStatement**: en ocasiones tenemos que ejecutar muchas sentencias SQL que son iguales, pero con distintos datos o parámetros. Por ejemplo: si tenemos que añadir 200 artículos en una tabla necesitaremos ejecutar 200 sentencias *INSERT* exactamente iguales, donde sólo varían los datos de cada artículo. La ejecución de cada uno de estos *INSERT* supone para el DBMS realizar las siguientes tareas:
 - Comprobar que la sentencia SQL es correcta.
 - Convertir los datos al tipo adecuado para el DBMS.
 - Ejecutar la sentencia SQL.

Realizar estas tareas 200 veces podría suponer un tiempo considerable. Para hacer más eficiente este proceso **PreparedStatement** nos brinda un sistema para precompilar la sentencia SQL y guardarla para ser ejecutada inmediatamente, sin necesidad de analizarla en cada caso.

PreparedStatement también nos permite añadir seguridad a nuestros programas, ya que podemos evitar "inyecciones SQL". [Pulsa aquí para conocer lo que son las inyecciones SQL.](#)

- **ResultSet:** los objetos *ResultSet* se crean como resultado de la ejecución de una sentencia *SELECT* y contienen el conjunto de registros resultado de su ejecución.

Primer proyecto con acceso a base de datos

Descargar el conector y crear el proyecto

En este apartado aprenderás a crear el proyecto Java que utilizaremos a lo largo de esta lección, descargarás de Internet el *driver* o conector para MySQL e importarás el *driver* dentro de dicho proyecto.

Antes de realizar el acceso desde Java a cualquier base de datos relacional a través de JDBC, es necesario realizar estas tres tareas:

- Descargar de Internet el *driver* o conector.
- Importar el *driver* o conector en el proyecto Java que deba acceder a la base de datos. Este *driver* será distinto en función del DBMS utilizado.
- Cargar en memoria el *driver*, ejecutando el método *Class.forName(...)* desde el código Java.

En este caso, vamos a ir directamente al paso 3 para ver qué ocurre. Crea un proyecto Java nuevo llamado *Unidad2* y, dentro la clase *Principal*, introduce el siguiente código:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        try {  
            Class.forName("com.mysql.jdbc.Driver");  
        } catch (ClassNotFoundException e) {  
            System.out.println("No se ha encontrado el driver para MySQL");  
            return;  
        }  
        System.out.println("Se ha cargado el Driver de MySQL");  
  
    }  
}
```

Class.forName("com.mysql.jdbc.Driver");

Con esta línea estamos cargando en memoria el *driver* de MySQL para Java. *Class.forName(...)* sirve para cargar en memoria un software específico, en este caso el *driver* o conector de MySQL. Pero no hemos descargado el *driver* ni lo hemos importado en el proyecto, por lo que ha saltado la excepción *ClassNotFoundException*, lo que ha provocado que el flujo del programa salte al bloque *catch(...)* y finalice ahí con la sentencia *return*.

Descargar el *driver* para MySQL

Conejero MySQL para Java

Pulsa en el botón de la derecha para descargar el *driver* conejero de MySQL para Java desde la web oficial de MySQL. <https://dev.mysql.com/downloads/connector/j/>

Localiza el área de descargas, moviéndote hacia abajo con la barra de desplazamiento si es necesario. El área de descargas tendrá un aspecto similar a éste:

The screenshot shows the MySQL Connector/J 5.1.46 download page. At the top, there are tabs for "Generally Available (GA) Releases" and "Development Releases". Below them, a dropdown menu says "Select Operating System: Platform Independent". There are two main download options:

- Platform Independent (Architecture Independent), Compressed TAR Archive**: Version 5.1.46, 4.2M. A "Download" button is shown.
- Platform Independent (Architecture Independent), ZIP Archive**: Version 5.1.46, 4.6M. A "Download" button is shown. This option is highlighted with a red box.

Below the download buttons, a note says: "We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download."

Aspecto del área de descargas.

Pulsa el botón "**Download**" para comenzar la descarga del archivo "**mysql-connector-java-5.1.46.zip**". Seguramente no comenzará directamente, sino que entrará en otra página similar a esta:

The screenshot shows the MySQL download page for the mysql-connector-java-5.1.46.zip file. The page has a header with the MySQL logo and navigation links for MySQL.COM, DOWNLOADS, DOCUMENTATION, and DEVELOPER ZONE. The DOWNLOADS link is underlined.

The main content area is titled "Begin Your Download" and shows the file name "mysql-connector-java-5.1.46.zip". It includes a note: "An Oracle Web Account provides you with the following advantages:" followed by a bulleted list. Below this is a box with "Login » using my Oracle Web account" and "Sign Up » for an Oracle Web account". At the bottom of the box, it says: "MySQL.com is using Oracle SSO for authentication. If you already have an Oracle Web account, click the Login link. Otherwise, you can signup for a free account by clicking the Sign Up link and following the instructions." A red box highlights the "No thanks, just start my download." link at the bottom.

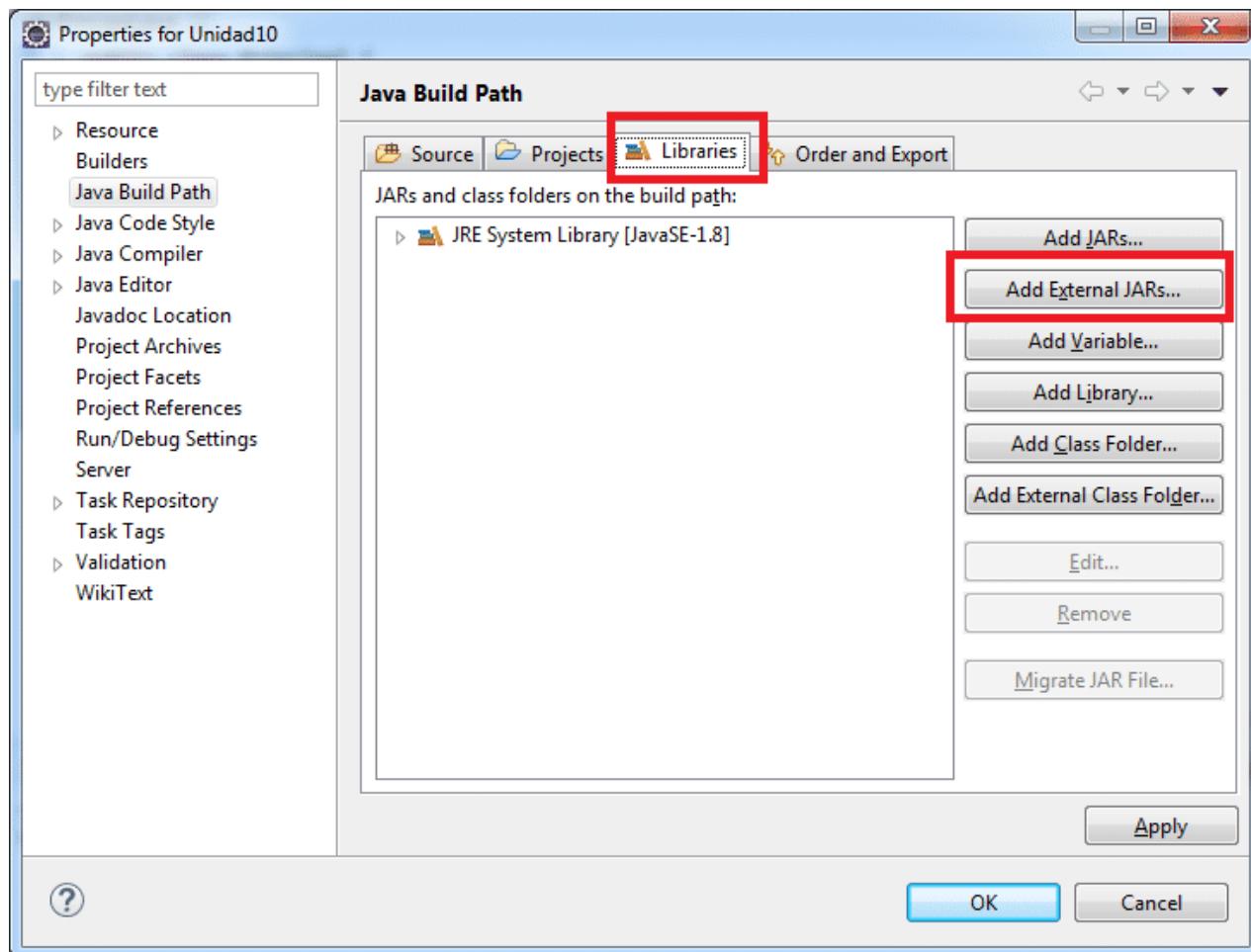
Descarga del conector de MySQL para Java.

Esta página te propone que te registres en la web, pero no es necesario, así que pulsa sobre el enlace "**No thanks, just start my download**". Cuando se haya terminado de descargar el archivo "mysql-connector-java-5.1.46.zip", descomprímelo en la ubicación que deseas para obtener una carpeta con nombre "**mysql-connector-java-5.1.46**".

Ya has completado el primer paso; la descarga del *driver*. Si trabajas con otro DBMS, el proceso no será muy distinto. Se trata de acceder a la web oficial del DBMS y buscar el enlace que permita descargar el *driver* para Java.

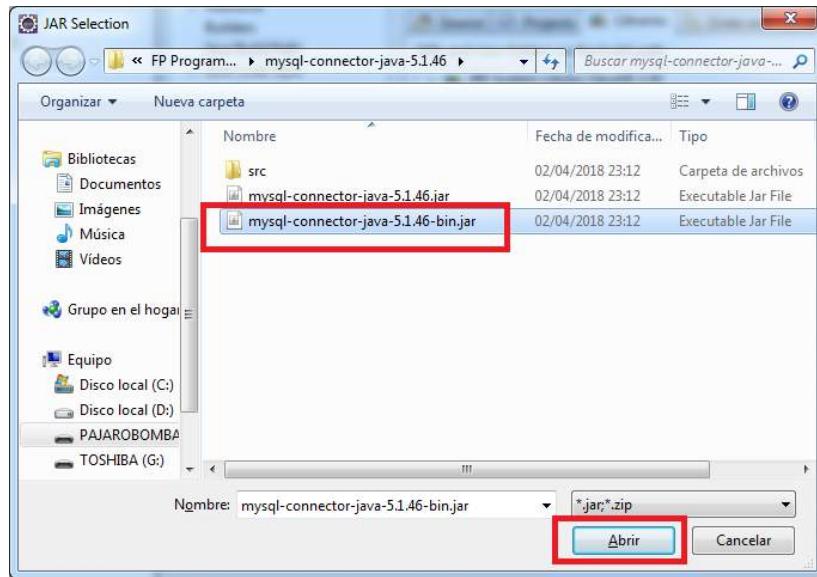
Importar el *driver* en el proyecto Java

Haz **clic derecho** sobre el nombre del proyecto *Unidad2* para obtener el menú contextual y selecciona la opción "**Properties**", que se encuentra al final del todo. Te encontrarás en el cuadro de diálogo "**Properties for Unidad2**".

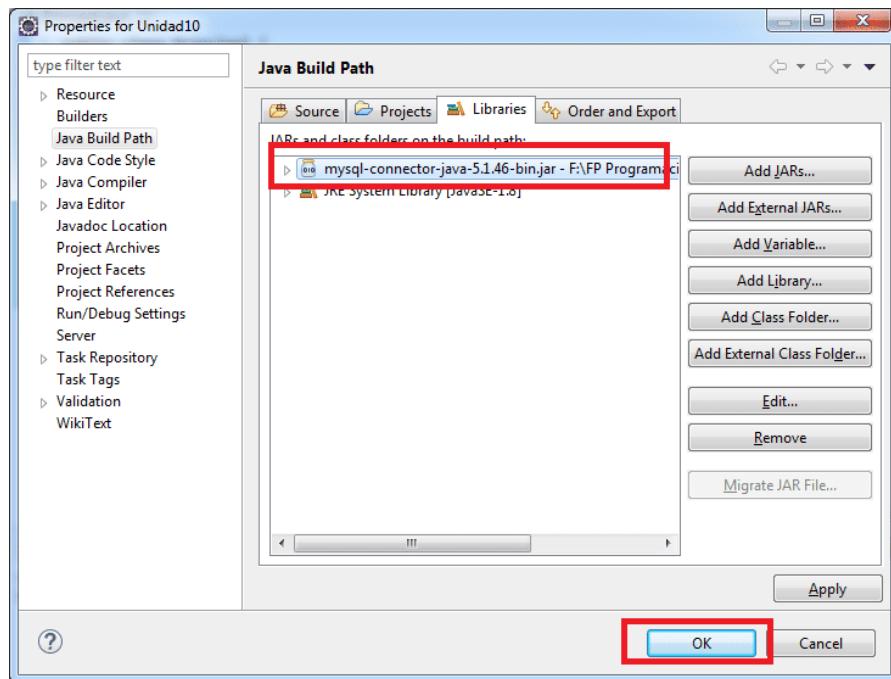


Project properties dialog.

Activa la ficha "**Libraries**" y haz clic en el botón "**Add External JARs...**" para abrir el cuadro de diálogo "**JAR Selection**", donde debes navegar a través del sistema de archivos de tu equipo hasta encontrar la carpeta que resultó de la descompresión del archivo .zip con el driver.

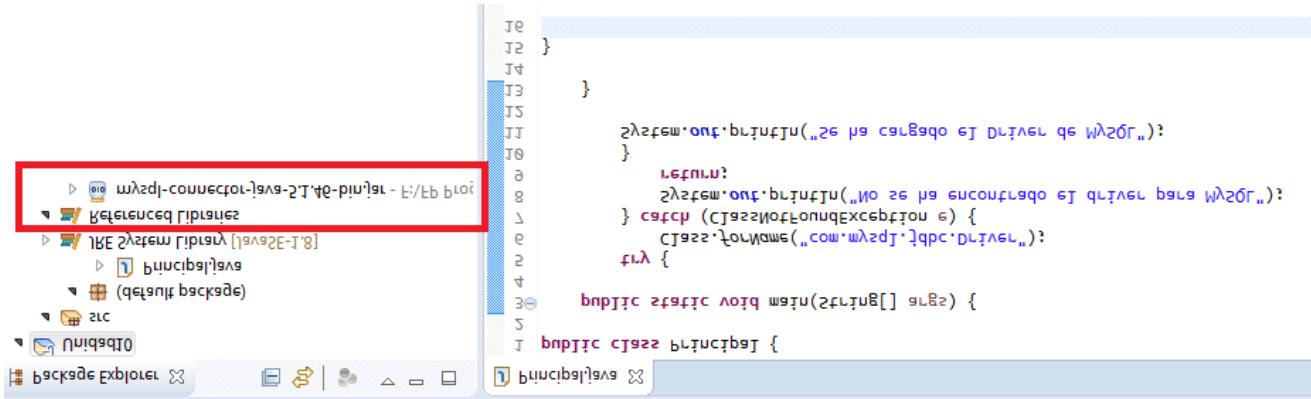


Selecciona el archivo "**mysql-connector-java-5.1.46-bin.jar**" y haz clic sobre el botón "**Abrir**". Vuelves así, de nuevo, al cuadro de diálogo "**Properties for Unidad2**", donde puedes comprobar que se ha incluido el archivo.



Aquí ya sólo tienes que pulsar el botón "**OK**" y el driver estará incluido en el proyecto.

Ahora, puedes observar de nuevo tu proyecto y ver que se ha incluido la carpeta "**Referenced Libraries**" con el nuevo driver.



```

    } }

    //Llegó la conexión("Se ha cargado el Driver de MySQL")?
    }

    //Ejecutar:
    //Llegó la conexión("No se ha encontrado conexión en la base de datos MySQL")?
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    //Proyecto: Sistemas de facturación
    //Código fuente: Ejemplo de conexión
    //Clase: ConnectionExample.java
}

```

Si ejecutas de nuevo el programa, comprobarás que, de momento, no hace gran cosa, sólo muestra el mensaje "Se ha cargado el Driver de MySQL". Pero, en realidad, hemos dado un paso muy importante: **nuestro programa ya está preparado para comenzar a trabajar con el API JDBC para comunicarse con MySQL.**

Conectarse con la base de datos

En este apartado vamos a establecer conexión con una base de datos MySQL llamada FERRETERIA.

Todos los ejemplos de código Java de esta unidad se conectarán a la base de datos FERRETERIA que creaste en la unidad anterior.

Para lograr conectar a la base de datos *FERRETERIA* utilizaremos las siguientes clases:

- ***DriverManager***: contiene información de todos los *drivers* o conectores cargados en memoria y actúa como proveedor, suministrando conexiones siempre y cuando tenga acceso a los *drivers* necesarios. Recuerda que hemos preparado el camino al *DriverManager* para que encuentre dichos *drivers* en el apartado anterior.
- ***Connection***: representa una conexión con una base de datos determinada y es suministrada por el *DriverManager*.

Para establecer conexión con una base de datos se utiliza el método estático ***getConnection()*** de la clase *DriverManager*, que retorna un objeto *Connection*. El método *getConnection()* tiene el siguiente formato:

```
Connection nombreObj = DriverManager.getConnection(String cadenaConexion, String
    usuario, String contraseña);
```

Formato del método *getConnection*.

El primer argumento es la **cadena de conexión**, que contendrá información sobre la base de datos a la que deseamos acceder. Los argumentos segundo y tercero contienen el nombre de usuario y contraseña del usuario autorizado a utilizar la base de datos.

La cadena de conexión

Una **cadena de conexión (connection string)** es un *String* que contiene información de acceso a una base de datos. El formato utilizado es diferente en función del proveedor de bases de datos. **JDBC** nos permite utilizar el mismo código para acceder a un gran número de formatos de bases de datos, cambiando solamente la cadena de conexión.

Puesto que los elementos y el formato de la cadena de conexión varían en función del proveedor de bases de datos, es difícil explicarlo de forma específica. En términos generales, **toda cadena de conexión se divide en las siguientes áreas:**

Proveedor:

Hace referencia al gestor de base de datos utilizado (*jdbc:sqlserver*, *jdbc:derby*, *jdbc:mysql*, etc.).

Dirección IP o nombre de servidor:

Se refiere al servidor donde está alojada la base de datos. Se utiliza en SGBD como SQL Server, Oracle o MySQL, que trabajan con tecnología cliente/servidor. Puede ser una DNS o una dirección IP.

Nombre de la instancia:

En SQL Server o MySQL un servidor puede contener varias instancias, cada instancia con sus propias bases de datos.

Número de puerto:

Necesario para establecer la conexión con el servidor. En MySQL es habitual que sea el 3306.

Nombre de la base de datos:

Nombre de la base a la que queremos acceder en nuestro programa.

Y ahora, pongámonos manos a la obra.

Completa el código de la clase *Principal* para establecer la conexión con la base de datos:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Principal {

    public static void main(String[] args) {
```

```
// Paso 1: Cargar el driver
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    System.out.println("No se ha encontrado el driver para MySQL");
    return;
}
System.out.println("Se ha cargado el Driver de MySQL");

// Paso 2: Establecer conexión con la base de datos
String cadenaConexion = "jdbc:mysql://localhost:3306/FERRETERIA";
String user = "root";
String pass = "amelia"; // sustituye por la contraseña que especificaste
durante la instalación de MySQL.
Connection con;
try {
    con = DriverManager.getConnection(cadenaConexion, user, pass);
} catch (SQLException e) {
    System.out.println("No se ha podido establecer la conexión con la
BD");
    System.out.println(e.getMessage());
    return;
}
System.out.println("Se ha establecido la conexión con la Base de datos");

// Paso 3: Interactuar con la BD (todavía pendiente)

// Paso 4: Cerrar la conexión
try {
    con.close();
} catch (SQLException e) {
    System.out.println("No se ha podido cerrar la conexión con la BD");
    System.out.println(e.getMessage());
    return;
}
System.out.println("Se ha cerrado la base de datos");
}

}
```

Establecemos conexión con la base de datos ejecutando la sentencia:

con = DriverManager.getConnection(cadenaConexion, user, pass);

Esta sentencia puede provocar una excepción de tipo ***SQLException*** si no se puede establecer la conexión por múltiples motivos (el usuario no está autorizado, la base de datos no existe, el servidor de BD está realizando tareas de mantenimiento, etc.).

Recuerda sustituir en la sentencia que sigue el valor "amelia" por el password que especificaste durante la instalación de MySQL.

String pass = "amelia";

Hemos utilizado la siguiente cadena de conexión:



Cadena de conexión para acceso a la base de datos MySQL FERRETERIA.

En este caso no hemos incluido nombre de instancia, ya que accedemos a un servidor MySQL que tiene una única instancia y no es necesario.

Como has podido comprobar con este ejemplo, cualquier programa típico que conecta a una base de datos se compone de cuatro partes:

- Cargar el *driver* en memoria.
- Establecer conexión con la base de datos, solicitando al *DriverManager* un objeto *Connection* y suministrando una cadena de conexión.
- Interactuar con la base de datos. Esta es la parte del programa que todavía tenemos pendiente.
- Cerrar la conexión con la base de datos.

Obtener un listado de clientes

En este apartado trabajaremos con las clases *Statement* y *ResultSet* para realizar un listado de clientes.

Vamos a completar el código de nuestra clase *Principal* y después lo analizaremos.

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Principal {

    public static void main(String[] args) {

        // Paso 1: Cargar el driver
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("No se ha encontrado el driver para MySQL");
            return;
        }
        System.out.println("Se ha cargado el Driver de MySQL");
    }
}
  
```

```

// Paso 2: Establecer conexión con la base de datos
String cadenaConexion = "jdbc:mysql://localhost:3306/FERRETERIA";
String user = "root";
String pass = "amelia";
Connection con;
try {
    con = DriverManager.getConnection(cadenaConexion, user, pass);
} catch (SQLException e) {
    System.out.println("No se ha podido establecer la conexión con la
BD");
    System.out.println(e.getMessage());
    return;
}
System.out.println("Se ha establecido la conexión con la Base de datos");

// Paso 3: Interactuar con la BD
try {
    Statement sentencia = con.createStatement();
    ResultSet rs = sentencia.executeQuery("SELECT * FROM CLIENTE");
    while (rs.next()) {
        System.out.print(rs.getString("NIF"));
        System.out.print(" - ");
        System.out.print(rs.getString("NOMBRE"));
        System.out.print(" - ");
        System.out.print(rs.getString("TLF"));
        System.out.println(); // Retorno de carro
    }
} catch (SQLException e) {
    System.out.println("Error al realizar el listado de productos");
    System.out.println(e.getMessage());
}

// Paso 4: Cerrar la conexión
try {
    con.close();
} catch (SQLException e) {
    System.out.println("No se ha podido cerrar la conexión con la BD");
    System.out.println(e.getMessage());
    return;
}
System.out.println("Se ha cerrado la base de datos");
}

}

```

Ahora, si ejecutas el programa obtendrás en pantalla un **listado de clientes**.

La parte del código que nos interesa analizar en este momento es la siguiente:

```

// Paso 3: Interactuar con la BD
try {
    Statement sentencia = con.createStatement();
    ResultSet rs = sentencia.executeQuery("SELECT * FROM CLIENTE");
    while (rs.next()) {
        System.out.print(rs.getString("NIF"));
        System.out.print(" - ");
        System.out.print(rs.getString("NOMBRE"));

```

```
        System.out.print(" - ");
        System.out.print(rs.getString("TLF"));
        System.out.println(); // Retorno de carro
    }
} catch (SQLException e) {
    System.out.println("Error al realizar el listado de productos");
    System.out.println(e.getMessage());
}
```

Por último, analicemos las **partes más importantes del programa:**

Statement sentencia = con.createStatement();

Será labor del objeto *Connection* (para nosotros la variable *con*) proveernos de un objeto de la clase *Statement* que nos servirá para ejecutar sentencias SQL. Llamaremos a nuestro objeto *Statement sentencia*.

ResultSet rs = sentencia.executeQuery("SELECT * FROM CLIENTE");

Los objetos *Statement* disponen del método *executeQuery()* para ejecutar sentencias de tipo *SELECT*, y retornan un objeto *ResultSet* con el conjunto de registros resultante. Dentro del programa, *rs* es nuestro objeto *ResultSet*. Para ejecutar otro tipo de sentencias, como *INSERT*, *UPDATE*, *DELETE*, etc., se utiliza el método *executeUpdate()*.

while (rs.next()) { }

El objeto *ResultSet* actúa como un cursor que permite desplazarse a través del conjunto de registros hacia delante, hacia atrás, al primero y al último. El método *next()* sitúa el puntero en el siguiente registro y retorna *true* si no es final de fichero, con lo que podemos utilizar su resultado para realizar un recorrido secuencial mientras siga retornando *true*.

System.out.print(rs.getString("NIF"));

El objeto *ResultSet* dispone de varios métodos para recuperar una columna y campo del registro activo: *getString* (para recuperar un campo de tipo cadena), *getInt* (para recuperar un campo de tipo Integer), *getFloat* (para recuperar un campo de tipo float), etc.

En función del tipo de datos con que se definió el campo en el DBMS, habrá que elegir un método u otro. Recuerda los tipos de datos de MySQL que estudiamos en la unidad anterior.

Estos métodos admiten como argumento una cadena con el nombre del campo, o bien un número entero, que representa el número de orden de la columna. La sentencia podría haberse escrito así: "System.out.print(rs.getString(1));" haciendo el 1 referencia a la primera columna, que es el *NIF*.

Añadir un nuevo cliente

Vamos a añadir un nuevo cliente a la tabla CLIENTE.

Puedes sustituir los datos que aparecen en el ejemplo por tus propios datos u otros inventados, y luego volver a ejecutar el ejemplo anterior del listado para comprobar que realmente se ha añadido el registro.

Comienza por añadir a tu proyecto la clase *PrincipalNuevo* con el código que podrás copiar y pegar desde aquí; luego, lo analizaremos paso a paso.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class PrincipalNuevo {

    public static void main(String[] args) {

        // Paso 1: Cargar el driver
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("No se ha encontrado el driver para MySQL");
            return;
        }
        System.out.println("Se ha cargado el Driver de MySQL");

        // Paso 2: Establecer conexión con la base de datos
        String cadenaConexion = "jdbc:mysql://localhost:3306/FERRETERIA";
        String user = "root";
        String pass = "amelia";
        Connection con;
        try {
            con = DriverManager.getConnection(cadenaConexion, user, pass);
        } catch (SQLException e) {
            System.out.println("No se ha podido establecer la conexión con la
BD");
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Se ha establecido la conexión con la Base de datos");

        // Paso 3: Interactuar con la BD
        try {
            String nif = "55667788A";
            String nombre = "DELGADO PEREZ CARLOS";
            String domicilio = "C/ ALENZA, 7";
            String tlf = "616667766";
            String ciudad = "MADRID";
            String sql = "INSERT INTO CLIENTE " +
                "(NIF, NOMBRE, DOMICILIO, TLF, CIUDAD) " +
                "VALUES ('" + nif + "', '" + nombre +
                "', '" + domicilio + "', '" + tlf + "', '" +
                ciudad + "')";
```

```

        System.out.println("Se va a ejecutar la siguiente sentencia SQL:");
        System.out.println(sql);
        Statement sentencia;
        sentencia = con.createStatement();
        int afectados = sentencia.executeUpdate(sql);
        System.out.println("Sentencia SQL ejecutada con éxito");
        System.out.println("Registros afectados: " + afectados);
    } catch (SQLException e) {
        System.out.println("Error al añadir nuevo cliente");
        System.out.println(e.getMessage());
    }

    // Paso 4: Cerrar la conexión
    try {
        con.close();
    } catch (SQLException e) {
        System.out.println("No se ha podido cerrar la conexión con la BD");
        System.out.println(e.getMessage());
        return;
    }
    System.out.println("Se ha cerrado la base de datos");
}

}

```

El objetivo de este programa es **enviar una orden al DBMS de MySQL para que ejecute la siguiente sentencia SQL:**

```
INSERT INTO CLIENTE (NIF, NOMBRE, DOMICILIO, TLF, CIUDAD) VALUES ('55667788A',
'DELGADO PEREZ CARLOS', 'C/ ALENZA, 7', '616667766', 'MADRID';
```

¡Ojo! Los valores de tipo texto en SQL deben ir encerrados entre comillas simples.

Los datos del nuevo cliente se encuentran en varias variables tipo *String*, y hemos tenido que montar la sentencia SQL concatenando los valores de dichas variables dentro de la sentencia y respetando la sintaxis requerida.

```

String nif = "55667788A";
String nombre = "DELGADO PEREZ CARLOS";
String domicilio = "C/ ALENZA, 7";
String tlf = "616667766";
String ciudad = "MADRID";
String sql = "INSERT INTO CLIENTE " +
            "(NIF, NOMBRE, DOMICILIO, TLF, CIUDAD) " +
            "VALUES ('" + nif + "', '" + nombre +
            "', '" + domicilio + "', '" + tlf + "', '" +
            ciudad + "');";

```

Una vez montada la sentencia SQL, la hemos ejecutado de nuevo con ayuda de un objeto *Statement*.

```
sentencia = con.createStatement();
int afectados = sentencia.executeUpdate(sql);
```

Pero esta vez, en lugar de utilizar el método *executeQuery(...)*, hemos utilizado el método ***executeUpdate(...)***, que **sirve para ejecutar una sentencia de actualización tipo *INSERT, UPDATE o DELETE*** y retorna el número de filas de la tabla que han sido afectadas, es decir, insertadas, borradas o modificadas.

Hemos añadido un solo cliente, pero recuerda que si tuviéramos que añadir 200 clientes mediante este sistema, tendríamos que ejecutar 200 sentencias iguales, donde sólo cambiarían los datos personales de las 200 personas. Por cada persona añadida, el DBMS tendría que comprobar la sintaxis de la sentencia, tarea que podría tener un coste considerable en tiempo de ejecución.

En el siguiente apartado realizaremos el mismo programa, pero usando la clase *PreparedStatement*, que nos permitirá mejorar considerablemente el rendimiento cuando ejecutemos la misma sentencia en numerosas ocasiones.

Usar PreparedStatement

PreparedStatement nos brinda un sistema para precompilar la sentencia SQL y guardarla para ser ejecutada inmediatamente, sin necesidad de analizarla en cada caso.

A continuación, expondremos el mismo programa del apartado anterior, pero usando la clase *PreparedStatement*:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class PrincipalNuevo {

    public static void main(String[] args) {

        // Paso 1: Cargar el driver
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("No se ha encontrado el driver para MySQL");
            return;
        }
        System.out.println("Se ha cargado el Driver de MySQL");
    }
}
```

```

// Paso 2: Establecer conexión con la base de datos
String cadenaConexion = "jdbc:mysql://localhost:3306/FERRETERIA";
String user = "root";
String pass = "amelia";
Connection con;
try {
    con = DriverManager.getConnection(cadenaConexion, user, pass);
} catch (SQLException e) {
    System.out.println("No se ha podido establecer la conexión con la
BD");
    System.out.println(e.getMessage());
    return;
}
System.out.println("Se ha establecido la conexión con la Base de datos");

// Paso 3: Interactuar con la BD
try {

    String sql = "INSERT INTO CLIENTE (NIF, NOMBRE, DOMICILIO, TLF,
CIUDAD) " +
                "VALUES (?, ?, ?, ?, ?)";

    String nif = "55667788A";
    String nombre = "DELGADO PEREZ CARLOS";
    String domicilio = "C/ ALENZA, 7";
    String tlf = "616667766";
    String ciudad = "MADRID";
    System.out.println("Se va a ejecutar la siguiente sentencia SQL:");
    System.out.println(sql);
    PreparedStatement sentencia;
    sentencia = con.prepareStatement(sql);
    sentencia.setString(1, nif);
    sentencia.setString(2, nombre);
    sentencia.setString(3, domicilio);
    sentencia.setString(4, tlf);
    sentencia.setString(5, ciudad);
    int afectados = sentencia.executeUpdate();
    System.out.println("Sentencia SQL ejecutada con éxito");
    System.out.println("Registros afectados: "+afectados);
} catch (SQLException e) {
    System.out.println("Error al añadir nuevo cliente");
    System.out.println(e.getMessage());
}

// Paso 4: Cerrar la conexión
try {
    con.close();
} catch (SQLException e) {
    System.out.println("No se ha podido cerrar la conexión con la BD");
    System.out.println(e.getMessage());
    return;
}
System.out.println("Se ha cerrado la base de datos");
}

}

```

Ahora vamos a analizar el código:

En primer lugar, *PreparedStatement* nos permite crear una sentencia SQL con parámetros que posteriormente podemos reemplazar:

```
String sql = "INSERT INTO CLIENTE (NIF, NOMBRE, DOMICILIO, TLF, CIUDAD) " +  
"VALUES (?, ?, ?, ?, ?);"
```

Cada interrogación es un parámetro que habrá que sustituir por los datos de un cliente. De esta forma nos sirve la misma cadena de texto para todas las sentencias del mismo tipo que vayamos a ejecutar.

```
PreparedStatement sentencia;  
sentencia = con.prepareStatement(sql);
```

Luego, hemos creado un objeto de tipo *PreparedStatement* asociado a la conexión abierta y lo hemos pasado como argumento al constructor la cadena SQL con los parámetros pendientes de reemplazar.

```
sentencia.setString(1, nif);  
sentencia.setString(2, nombre);  
sentencia.setString(3, domicilio);  
sentencia.setString(4, tlf);  
sentencia.setString(5, ciudad);
```

Por cada una de las interrogaciones (parámetros) dentro de la cadena SQL, tenemos que ejecutar un método *set...* con el siguiente formato:

```
obj.setxxx(ordenParametro, valorParametro);
```

Donde xxx hace referencia al tipo de datos (*setInt(...)*, *setString(...)*, *setFloat(...)*, *setDouble(...)*, etc.).

```
int afectados = sentencia.executeUpdate();
```

El último paso es ejecutar la sentencia con los métodos *executeQuery()* o *executeUpdate()*.

Borrar un cliente

En el siguiente apartado ejecutaremos una sentencia SQL de eliminación, es decir, una sentencia *DELETE*.

Eliminaremos de la tabla *CLIENTE* el registro previamente añadido y utilizaremos de nuevo la clase *PreparedStatement*.

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class PrincipalBorrar {

    public static void main(String[] args) {

        // Paso 1: Cargar el driver
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("No se ha encontrado el driver para MySQL");
            return;
        }
        System.out.println("Se ha cargado el Driver de MySQL");

        // Paso 2: Establecer conexión con la base de datos
        String cadenaConexion = "jdbc:mysql://localhost:3306/FERRETERIA";
        String user = "root";
        String pass = "amelia";
        Connection con;
        try {
            con = DriverManager.getConnection(cadenaConexion, user, pass);
        } catch (SQLException e) {
            System.out.println("No se ha podido establecer la conexión con la
BD");
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Se ha establecido la conexión con la Base de datos");

        // Paso 3: Interactuar con la BD
        try {

            String sql = "DELETE FROM CLIENTE WHERE NIF=?";

            String nif = "55667788A";
            System.out.println("Se va a ejecutar la siguiente sentencia SQL:");
            System.out.println(sql);
            PreparedStatement sentencia;
            sentencia = con.prepareStatement(sql);
            sentencia.setString(1, nif);
            int afectados = sentencia.executeUpdate();
            System.out.println("Sentencia SQL ejecutada con éxito");
            System.out.println("Registros afectados: "+afectados);
        } catch (SQLException e) {
            System.out.println("Error al borrar el cliente");
            System.out.println(e.getMessage());
        }

        // Paso 4: Cerrar la conexión
        try {
            con.close();
        } catch (SQLException e) {
    
```

```

        System.out.println("No se ha podido cerrar la conexión con la BD");
        System.out.println(e.getMessage());
        return;
    }
    System.out.println("Se ha cerrado la base de datos");

}
}

```

Vamos a analizar las partes más importante del código:

- Hemos creado una cadena de texto SQL con un parámetro que recogerá el NIF del cliente que queremos borrar:
`String sql = "DELETE FROM CLIENTE WHERE NIF=?";`
- Hemos creado un objeto *PreparedStatement* asociado con la conexión abierta, pasando como argumento al constructor la sentencia SQL con el parámetro:
`String nif = "55667788A";`
`PreparedStatement sentencia;`
`sentencia = con.prepareStatement(sql);`
- Hemos asignado valor al parámetro ejecutando la sentencia:
`sentencia.setString(1, nif);`
- Hemos ejecutado la sentencia SQL con la siguiente línea de código:
`int afectados = sentencia.executeUpdate();`

Programar la gestión de clientes

Vamos a ver un ejemplo completo de gestión de la tabla CLIENTE a partir de un menú que sirve como interfaz para el usuario.

Puedes crear un nuevo proyecto Java llamado **GestionClientes**, donde añadirás la siguiente clase:

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

public class Principal {
    private static Scanner lector;

    public static void main(String[] args) {

        Connection con = abrirConexionBD();
        if (con == null)
            return;
    }
}

```

```

lector = new Scanner(System.in);
int opc;

do {
    mostrarMenu();
    opc = lector.nextInt();
    lector.nextLine(); // Recogemos el retorno de carro.
    System.out.println();
    switch (opc) {
        case 1:
            listarClientes(con);
            break;
        case 2:
            nuevoCliente(con);
            break;
        case 3:
            borrarCliente(con);
            break;
        case 4:
            modificarCliente(con);
            break;
        case 5:
            buscarCliente(con);
            break;
        case 6:
            System.out.println("Hasta pronto");
            break;
        default:
            System.out.println("Opción incorrecta");
    }
} while (opc != 6);

cerrarConexion(con); // Pasamos como argumento la conexión a cerrar.
lector.close();
}

private static Connection abrirConexionBD() {
    Connection con;
    // Paso 1: Cargar el driver
    try {
        Class.forName("com.mysql.jdbc.Driver");
    } catch (ClassNotFoundException e) {
        System.out.println("No se ha encontrado el driver para MySQL");
        return null;
    }
    System.out.println("Se ha cargado el Driver de MySQL");

    // Paso 2: Establecer conexión con la base de datos
    String cadenaConexion = "jdbc:mysql://localhost:3306/FERRETERIA";
    String user = "root";
    String pass = "amelia";
    try {
        con = DriverManager.getConnection(cadenaConexion, user, pass);
    } catch (SQLException e) {
        System.out.println("No se ha podido establecer la conexión con la
BD");
        System.out.println(e.getMessage());
        return null;
    }
    System.out.println("Se ha establecido la conexión con la Base de datos");
    return con; // Devolvemos la conexión abierta.
}

```

```

private static void cerrarConexion(Connection con) {
    try {
        con.close();
    } catch (SQLException e) {
        System.out.println("No se ha podido cerrar la conexión con la BD");
        System.out.println(e.getMessage());
        return;
    }
    System.out.println("Se ha cerrado la base de datos");
}

public static void mostrarMenu() {
    System.out.println();
    System.out.println("GESTION DE CLIENTES");
    System.out.println("-----");
    System.out.println("1. Listado de clientes");
    System.out.println("2. Añadir nuevo cliente");
    System.out.println("3. Borrar cliente");
    System.out.println("4. Modificar dirección y teléfono");
    System.out.println("5. Buscar cliente");
    System.out.println("6. Terminar programa");
    System.out.println("-----");
    System.out.println("¿Qué opción eliges?");
}

private static void listarClientes(Connection con) {
    try {
        Statement sentencia = con.createStatement();
        ResultSet rs = sentencia.executeQuery("SELECT * FROM CLIENTE");
        while (rs.next()) {
            System.out.print(rs.getString(1));
            System.out.print(" - ");
            System.out.print(rs.getString("NOMBRE"));
            System.out.print(" - ");
            System.out.print(rs.getString("DOMICILIO"));
            System.out.print(" - ");
            System.out.print(rs.getString("TLF"));
            System.out.println(); // Retorno de carro
        }
    } catch (SQLException e) {
        System.out.println("Error al realizar el listado de productos");
        System.out.println(e.getMessage());
    }
}

private static void nuevoCliente(Connection con) {
    try {
        String sql = "INSERT INTO CLIENTE (NIF, NOMBRE, DOMICILIO, TLF,
CIUDAD) " + "VALUES (?, ?, ?, ?, ?, ?)";
        System.out.println("Introduce NIF del nuevo cliente: ");
        String nif = lector.nextLine();
        System.out.println("Introduce nombre: ");
        String nombre = lector.nextLine();
        System.out.println("Introduce dirección: ");
        String domicilio = lector.nextLine();
        System.out.println("Introduce teléfono: ");
        String tlf = lector.nextLine();
        System.out.println("Introduce ciudad: ");
        String ciudad = lector.nextLine();
        System.out.println("Se va a ejecutar la siguiente sentencia SQL:");
        System.out.println(sql);
    }
}

```

```

        PreparedStatement sentencia;
        sentencia = con.prepareStatement(sql);
        sentencia.setString(1, nif);
        sentencia.setString(2, nombre);
        sentencia.setString(3, domicilio);
        sentencia.setString(4, tlf);
        sentencia.setString(5, ciudad);
        int afectados = sentencia.executeUpdate();
        System.out.println("Sentencia SQL ejecutada con éxito");
        System.out.println("Registros afectados: " + afectados);
    } catch (SQLException e) {
        System.out.println("Error al añadir nuevo cliente");
        System.out.println(e.getMessage());
    }
}

private static void borrarCliente(Connection con) {
    String sql = "DELETE FROM CLIENTE WHERE NIF=?";
    System.out.println("Escribe NIF del cliente a borrar: ");
    String nif = lector.nextLine();
    PreparedStatement sentencia;
    int afectados;
    try {
        sentencia = con.prepareStatement(sql);
        sentencia.setString(1, nif);
        afectados = sentencia.executeUpdate();
    } catch (SQLException e) {
        System.out.println("No se ha podido borrar el cliente");
        System.out.println(e.getMessage());
        return;
    }
    System.out.println("Sentencia SQL ejecutada con éxito");
    System.out.println("Registros afectados: "+afectados);
}

private static void modificarCliente(Connection con) {
    String sql = "UPDATE CLIENTE SET DOMICILIO=?, TLF=? WHERE NIF=?;";
    System.out.println("Escribe NIF del cliente a modificar: ");
    String nif = lector.nextLine();
    System.out.println("Escribe nueva dirección: ");
    String domicilio = lector.nextLine();
    System.out.println("Escribe nuevo teléfono: ");
    String tlf = lector.nextLine();
    PreparedStatement sentencia;
    int afectados;

    try {
        sentencia = con.prepareStatement(sql);
        sentencia.setString(1, domicilio);
        sentencia.setString(2, tlf);
        sentencia.setString(3, nif);
        afectados = sentencia.executeUpdate();
    } catch (SQLException e) {
        System.out.println("No se ha podido modificar el cliente");
        System.out.println(e.getMessage());
        return;
    }
    System.out.println("Sentencia SQL ejecutada con éxito");
    System.out.println("Registros afectados: "+afectados);
}

private static void buscarCliente(Connection con) {
}

```

```

        try {
            Statement sentencia = con.createStatement();
            System.out.println("Escribe nombre del cliente buscado: ");
            String nombre = lector.nextLine();
            ResultSet rs = sentencia.executeQuery("SELECT * FROM CLIENTE WHERE
NOMBRE LIKE '%" + nombre + "%'");
            while (rs.next()) {
                System.out.print(rs.getString("NIF"));
                System.out.print(" - ");
                System.out.print(rs.getString("NOMBRE"));
                System.out.print(" - ");
                System.out.print(rs.getString("DOMICILIO"));
                System.out.print(" - ");
                System.out.print(rs.getString("TLF"));
                System.out.println(); // Retorno de carro
            }
        } catch (SQLException e) {
            System.out.println("Error al realizar el listado de productos");
            System.out.println(e.getMessage());
        }
    }
}

```

No olvides que, para que funcione, debes importar el *driver* de MySQL dentro del proyecto.

El programa realiza las siguientes tareas:

1. En primer lugar, abre la base de datos llamando al método estático ***abrirConexionBD()***, que devuelve la conexión y deja la base de datos abierta.
2. Una vez abierta la base de datos, el programa permite al usuario realizar las operaciones deseadas, hasta que decida terminar pulsando la opción 6 en el siguiente **menú**, que sirve de interfaz:

```

GESTION DE CLIENTES
-----
1. Listado de clientes
2. Añadir nuevo cliente
3. Borrar cliente
4. Modificar dirección y teléfono
5. Buscar cliente
6. Terminar programa
-----
¿Qué opción eliges?

```

Según las opciones que vaya seleccionando el cliente, el programa llama a diversos métodos que realizan cada tarea.

3. Por último, se cierra la base de datos llamando al método ***cerrarConexion(con)***, que recibe como argumento el objeto *Connection* a cerrar.

Como ves, el usuario puede realizar varias operaciones con la base de datos, pero **la apertura y cierre de la conexión se realizan una única vez**.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

1. El **API JDBC** provee un mecanismo sencillo para acceder a bases de datos relacionales desde aplicaciones Java. Se encuentra disponible en el paquete **java.sql**, que pertenece a la “JRE System Library”, por lo que no es necesario cargar ninguna librería externa. JDBC se sitúa como una capa dentro de nuestra aplicación Java capaz de comunicarse con el DBMS.
2. La clase **DriverManager** se encarga de gestionar los *drivers* de los distintos proveedores de BD y de proveer al programa del objeto **Connection** para establecer la conexión con la BD.
3. La clase **Connection** representa una conexión con una base de datos.
4. La clase **Statement** permite ejecutar sentencias SQL sobre una conexión a una BD abierta.
5. **PreparedStatement** nos brinda un sistema para precompilar la sentencia SQL y guardarla para ser ejecutada inmediatamente, sin necesidad de analizarla en cada caso.
6. Los objetos de tipo **ResultSet** se crean como resultado de la ejecución de una sentencia **SELECT** y contienen el conjunto de registros resultado de su ejecución.

MP_0486. Acceso a datos

**UF2. Manejo de conectores a bases
de datos relacionales**

2.4. Gestión de transacciones



Índice

Objetivos	3
Transacciones.....	4
Concepto	4
Propiedades ACID.....	4
Control de transacciones en MySQL	5
Estados de una transacción	6
Transacciones en Java.....	7
Aplicación práctica.....	7
Despedida	10
Resumen.....	10

Objetivos

En esta unidad perseguimos los siguientes objetivos:

- Conocer el concepto de transacción.
- Realizar programas Java con el API JDBC que agrupen un conjunto de operaciones de actualización de la base de datos dentro de una transacción.

Transacciones

Concepto

Una transacción es el conjunto de operaciones necesarias para llevar a cabo una determinada tarea que implique modificaciones en la base de datos.

Por ejemplo, para realizar una venta serán necesarias las siguientes operaciones en la base de datos:

- Añadir un nuevo registro con los datos del cliente, si no está registrado en el sistema.
- Añadir la nueva factura.
- Añadir las líneas de detalle de la venta.
- Actualizar el stock en el almacén.

Todas estas operaciones **deben ser tratadas dentro de una unidad de trabajo, de modo que se realicen todas las operaciones o ninguna**. Sería un desastre que la tarea se quedara a medias por un error; por ejemplo, que se añadiera la factura y no se actualizara el stock.

Para garantizar que esto no ocurra, todo el bloque de operaciones ha de estar incluido dentro de una transacción.

Propiedades ACID

Una transacción debe cumplir con un grupo de propiedades denominadas “**propiedades ACID**”.

Las siglas ACID corresponden a las iniciales de:

Atomicidad

La transacción debe ser completa, es decir, **se ejecutarán todas las operaciones o ninguna**.

Consistencia

La ejecución de la transacción **no debe romper las reglas de integridad**. Por ejemplo: la transacción no debe provocar que se agregue una factura sin cliente relacionado, o una línea de detalle que no pertenezca a ninguna factura.

Aislamiento

La ejecución de **una transacción no debe afectar al funcionamiento de otra transacción**. El SGBD debe garantizar una buena gestión de bloqueo de registros cuando sea necesario.

Durabilidad

Es decir, la **persistencia (permanencia en el tiempo) de los nuevos datos**.

Un SGBD que quiera llamarse transaccional debe cumplir las cuatro propiedades ACID obligatoriamente .

Control de transacciones en MySQL

La gestión de transacciones en MySQL se realiza a partir de estas tres sentencias:

BEGIN

Inicio de la transacción.

COMMIT

Finalización de la transacción.

ROLLBACK

Aborta la transacción revirtiendo los cambios.

Para agrupar un conjunto de operaciones dentro de una transacción hay que encerrarlas en un **bloque BEGIN ... COMMIT**.

```
USE FERRETERIA;

BEGIN;
    INSERT INTO CLIENTE VALUES('12345678A','LUIS LOPEZ LOPEZ', 'C/ SOL, 3','639-639-639', 'MADRID');
    INSERT INTO FACTURA VALUES(5446,'2017-12-15',false,'12345678A');
    INSERT INTO DETALLE VALUES(5446,'MAR1',1,13);
    INSERT INTO DETALLE VALUES(5446,'TOR7',2,0.9);
    UPDATE PRODUCTO SET STOCK=STOCK-1 WHERE CODIGO='MAR1';
    UPDATE PRODUCTO SET STOCK=STOCK-2 WHERE CODIGO='TOR7';
COMMIT;
```

Ahora vamos a realizar una transacción similar para otro cliente y otra factura. Pero esta vez la abortaremos en el último momento.

```
BEGIN;
    INSERT INTO CLIENTE VALUES ('23456789A', 'ROSA LOPEZ LOPEZ', 'C/ LUNA, 5', '612-612-612', 'MADRID');
    INSERT INTO FACTURA VALUES (5447, '2017-12-15', false, '23456789A');
    INSERT INTO DETALLE VALUES (5447, 'MAR1', 1, 13);
    INSERT INTO DETALLE VALUES (5447, 'TOR7', 2, 0.9);
    UPDATE PRODUCTO SET STOCK=STOCK-1 WHERE CODIGO='MAR1';
    UPDATE PRODCTO SET STOCK=STOCK-2 WHERE CODIGO='TOR7';
    SELECT * FROM DETALLE;
ROLLBACK;
```

Dentro del código de la transacción hemos colocado una sentencia `SELECT * FROM DETALLE`. Verás que la sentencia ha devuelto todas las líneas de detalle, incluidas las de la nueva factura (5447). Ahora, ejecuta de nuevo la siguiente sentencia:

```
SELECT * FROM DETALLE;
```

Verás que ahora no aparecen los detalles de la nueva factura, porque después del `ROLLBACK` los cambios han sido revertidos.

IMPORTANTE: si inicias una transacción (`BEGIN`) y luego no haces el `COMMIT`, la transacción terminará siendo rechazada, como si hubieras hecho un `ROLLBACK`.

Estados de una transacción

Las transacciones pueden encontrarse en uno de los siguientes estados:

- **Activa:** la transacción está activa durante todo el tiempo que dura su ejecución.
- **Parcialmente comprometida:** se acaba de realizar la última instrucción de la transacción, pero todavía no se ha hecho `COMMIT TRAN` (la finalización de la transacción).
- **Fallida:** ha ocurrido algún fallo en alguna de las operaciones de la transacción y la transacción no puede finalizar. Después de pasar por este estado, pasará por el estado de *abortada*.
- **Abortada:** la transacción pasa a este estado cuando se ha restablecido la BD a su estado anterior. En este caso, se deshacen los cambios realizados como si no hubiese ocurrido nada.
- **Comprometida:** cuando todas las operaciones se han completado con éxito y la BD ha quedado actualizada.

Transacciones en Java

Aplicación práctica

En este apartado pondremos en práctica el uso de transacciones desde un programa Java.
El siguiente programa dará de alta una nueva factura en la base de datos FERRETERIA. Dicha tarea implica **dar de alta al cliente, añadir la factura, añadir las líneas de detalle y actualizar el stock de los productos vendidos.**

Para empezar, crea un nuevo proyecto en eclipse, y copia y pega este código para la clase *Principal*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class Principal {

    public static void main(String[] args) throws SQLException {

        // Paso 1: Cargar el driver
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("No se ha encontrado el driver para MySQL");
            return;
        }
        System.out.println("Se ha cargado el Driver de MySQL");

        // Paso 2: Establecer conexión con la base de datos
        String cadenaConexion = "jdbc:mysql://localhost:3306/Ferreteria";
        String user = "root";
        String pass = "amelia";
        Connection con;
        try {
            con = DriverManager.getConnection(cadenaConexion, user,
pass);
        } catch (SQLException e) {
            System.out.println("No se ha podido establecer la conexión con la BD");
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Se ha establecido la conexión con la Base de datos");

        // Paso 3: Interactuar con la BD
        try {
            con.setAutoCommit(false);
            Statement sentencia = con.createStatement();
        }
    }
}
```

```
String sql = "INSERT INTO CLIENTE VALUES  
('51666666A', 'ROCAFLORE DELGADO RODOLFO', 'C/ PITONISA, 45', '616656644',  
'SEVILLA');";  
sentencia.executeUpdate(sql);  
sql = "INSERT INTO FACTURA VALUES (5446, '2018/04/23', 0,  
'51666666A');";  
sentencia.executeUpdate(sql);  
sql = "INSERT INTO DETALLE VALUES (5446, 'MAR2', 1, 7)";  
sentencia.executeUpdate(sql);  
sql = "INSERT INTO DETALLE VALUES (5446, 'TOR7', 2,  
0.8)";  
sentencia.executeUpdate(sql);  
sql = "UPDATE PRODUCTO SET STOCK=STOCK-1 WHERE  
CODIGO='MAR2'";  
sentencia.executeUpdate(sql);  
sql = "UPDATE PRODUCTO SET STOCK=STOCKKK-2 WHERE  
CODIGO='TOR7'";  
sentencia.executeUpdate(sql);  
  
con.commit();  
  
} catch (SQLException e) {  
    System.out.println("Ha ocurrido un error al añadir la  
factura");  
    System.out.println(e.getMessage());  
    con.rollback();  
}  
  
// Paso 4: Cerrar la conexión  
try {  
    con.close();  
} catch (SQLException e) {  
    System.out.println("No se ha podido cerrar la conexión  
con la BD");  
    System.out.println(e.getMessage());  
    return;  
}  
System.out.println("Se ha cerrado la base de datos");  
}  
}
```

Este programa ejecuta seis sentencias de manipulación de datos, cuatro inserciones y dos actualizaciones. Sin embargo, observa que en la última sentencia SQL la palabra *STOCK*, está mal escrita, provocando así una excepción que llevará el control del programa al bloque *catch*, donde se aborta la transacción con un método *rollback()*.

En realidad, ninguna modificación ha sido efectuada a pesar de que las cinco primeras sentencias fueron correctas. **La gestión de la transacción está garantizando que se ejecuten todas las sentencias o ninguna.**

IMPORTANTE: no te olvides de importar el driver de MySQL para Java.

Ahora, vamos a analizar más detenidamente el código anterior:

con.setAutoCommit(false);

El método `setAutoCommit()` de la clase `Connection` permite establecer el valor de la propiedad `autoCommit` para especificar cuándo será completada una transacción. Esta propiedad tiene por defecto el valor `true`, lo que significa que cada instrucción SQL se ejecuta y confirma en una transacción individual. Por lo tanto, mientras la propiedad `autoCommit` tenga el valor `true`, no será posible agrupar varias sentencias SQL en una única transacción.

Por ello, necesitamos comenzar la transacción estableciendo el valor `false` a la propiedad `autoCommit`.

con.commit();

El método `commit()` finaliza la transacción, realizando físicamente todas las actualizaciones especificadas por la colección de sentencias SQL que se han ido ejecutando.

con.rollback();

Aborta la transacción revirtiendo todos los cambios efectuados por el grupo de sentencias SQL que se han ejecutado dentro de la transacción. Se utiliza el método `rollback()` como respuesta a una situación de excepción.

Si has ejecutado el programa, habrás comprobado que la transacción ha funcionado como esperábamos y ninguna de las modificaciones se ha llegado a efectuar.

Ahora, prueba a corregir la palabra `STOCK` en la última sentencia SQL para que puedas comprobar después si se efectuaron todas las modificaciones, añadiéndose la nueva factura.

Tendrás que utilizar el entorno de MySQL Workbench para comprobar que se han guardado todos los datos.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Una **transacción** es el conjunto de operaciones necesarias para llevar a cabo una determinada tarea que implica modificaciones en la base de datos. Un SGBD, para llamarse transaccional, debe garantizar las propiedades **ACID** (Atomicidad, Consistencia, Aislamiento, Durabilidad) en las transacciones.
- Java trata cada instrucción SQL dentro de una transacción individual de manera predeterminada. Para agrupar varias instrucciones en una sola transacción hay que comenzar por cambiar dicho comportamiento, ejecutando la siguiente sentencia:

```
con.setAutoCommit(false);
```

Siendo *con* el objeto *Connection*.

- El método *commit()* de la clase *Connection* finaliza la transacción, permitiendo así la actualización de la base de datos con todos los cambios especificados por las sentencias SQL ejecutadas.
- El método *rollback()* de la clase *Connection* aborta la transacción.

3.1. Conceptos básicos sobre mapeo objeto-relacional. Frameworks ORM.



Índice

Introducción	3
Objetivos.....	3
Conceptos previos	4
Los JavaBeans.....	4
¿Qué es el mapeo objeto-relacional?	5
Las clases de entidad	7
Las anotaciones.....	9
¿Qué son los frameworks ORM?.....	13
Despedida	15
Resumen.....	15

Introducción

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Sentar algunos conceptos previos para comprender mejor el funcionamiento de los *frameworks ORM*: beans y anotaciones.
- Describir el significado de *framework ORM* y la técnica ORM (mapeo objeto relacional).
- Enumerar algunos frameworks comerciales que implementan el ORM.
- Describir el funcionamiento de las herramientas ORM.

Conceptos previos

Los JavaBeans

Un *JavaBean* es una clase que debe cumplir una serie de condiciones.

Estas condiciones son:

- Debe implementar la interfaz *Serializable*, lo que otorga a sus objetos la capacidad de persistencia.
- Debe tener un constructor vacío (que no reciba argumentos), aunque luego puede tener otros constructores. De esta forma se podrán crear objetos estándar.
- Todas las propiedades del objeto serán privadas y accesibles mediante métodos *get/set* que serán públicos.
- Para los métodos *get/set*, hay que seguir cuidadosamente la nomenclatura estándar. Para una propiedad privada llamada *precio*, los métodos *get/set* serán *getPrecio* y *setPrecio*. Las palabras *get* y *set* en minúscula y el nombre de la propiedad con la primera letra mayúscula.

El concepto *JavaBean* fue creado por Sun Microsystems, aunque luego esta compañía fue adquirida por Oracle en 2010. Los *JavaBean* nacieron con el objetivo de ser utilizados como componentes software reutilizables.

Veamos un ejemplo de clase que cumple con las especificaciones de los *JavaBeans*.

```
import java.io.Serializable;
import java.time.LocalDateTime;

public class Llamada implements Serializable {
    private static final long serialVersionUID = 6164080316086841480L;

    private LocalDateTime fechaHora;
    private String emisor; // Nombre de la persona que llamo.
    private String motivo; // Motivo de la llamada.

    public Llamada() {
        this.fechaHora = LocalDateTime.now();
    }

    public LocalDateTime getFechaHora() {
        return fechaHora;
    }
    public void setFechaHora(LocalDateTime fechaHora) {
        this.fechaHora = fechaHora;
    }
    public String getEmisor() {
        return emisor;
    }
}
```

```
    }
    public void setEmisor(String emisor) {
        this.emisor = emisor;
    }
    public String getMotivo() {
        return motivo;
    }
    public void setMotivo(String motivo) {
        this.motivo = motivo;
    }

    @Override
    public String toString() {
        return this.emisor + " llamo el " + this.fechaHora + " para " +
this.motivo;
    }
}
```

La clase *Llamada* cumple con la especificación JavaBeans porque es serializable, tiene un constructor sin argumentos y sus métodos son privados y accesibles mediante métodos *get/set*, generados cumpliendo la nomenclatura estándar.

Podríamos construir un objeto en una clase con método *main()* de la siguiente forma:

```
public class Principal {

    public static void main(String[] args) {
        Llamada unaLlamada = new Llamada();
        unaLlamada.setEmisor("Carlos Pérez");
        unaLlamada.setMotivo("Pedir información");
        System.out.println(unaLlamada); // Invoca al método toString();
    }
}
```

¿Qué es el mapeo objeto-relacional?

En unidades anteriores aprendimos cómo crear programas Java capaces de comunicarse con una base de datos relacional a través del API JDBC.

Ahora vamos a dar un paso más, aprendiendo a utilizar otra técnica muy empleada en aplicaciones empresariales Java: **el Mapeo Objeto-Relacional**.

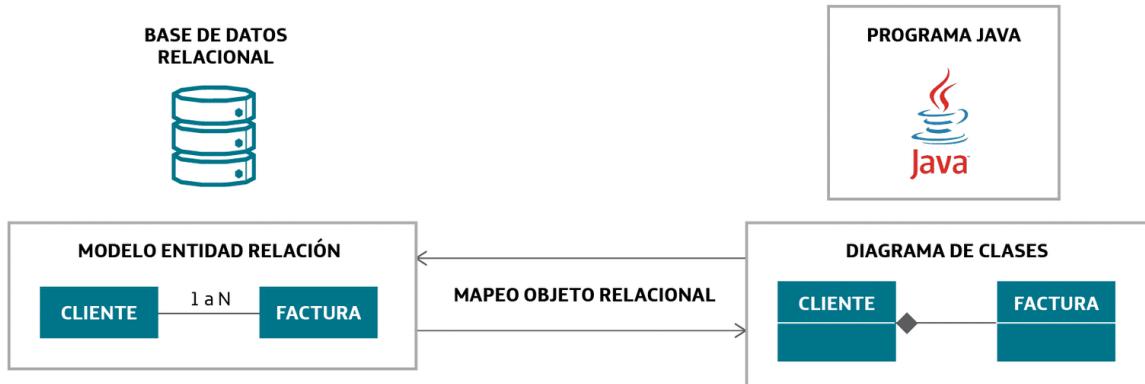
Una base de datos relacional está compuesta por un conjunto de tablas asociadas entre sí, donde cada tabla tiene una clave principal. Por otro lado, un programa Java funciona mediante un conjunto de objetos que se comunican entre sí para lograr un determinado objetivo.

¿Hay similitudes entre ambas cosas?

BD = conjunto de tablas asociadas entre ellas.
Programa java = conjunto de objetos relacionados entre ellos.

El "mapeo objeto-relacional" pretende acercar el mundo de las bases de datos al mundo de los objetos Java.

Para ello se parte del Modelo Entidad-Relación de la base de datos relacional y, a través del proceso de "Mapeo Objeto-Relacional" u ORM, se construye una base de datos *Orientada a Objetos virtual*, es decir, en memoria. En dicho proceso, para cada entidad física o registro de la BD con la que tengamos que trabajar, existirá su correspondiente objeto Java con la misma estructura, de modo que programaremos de una forma más cercana a la filosofía de clases y objetos de Java.



Por un lado, tenemos una base de datos con su estructura relacional, con sus tablas físicas y sus asociaciones, y por otro lado, tenemos clases Java relacionadas que imitan la estructura de la base de datos.

Será una herramienta denominada **Framework ORM** la encargada de mapear las clases Java con las tablas físicas de base de datos.

Pero, ¿qué significa *mapear*?

Pensando en nuestra base de datos FERRETERIA, mapear un cliente correspondería a crear automáticamente un objeto Java de la clase *Cliente* con sus objetos *Factura* asociados, a través de la lectura de un registro de la tabla *CLIENTE* de la base de datos *FERRETERIA*. Este proceso lo realiza automáticamente el *framework ORM*.

También podemos crear un nuevo objeto *Cliente*, con sus objetos *Factura* asociados si es necesario, y a continuación hacerlo persistir, es decir, guardarlo de forma física en la base de datos.

Las clases de entidad

Hemos visto cómo en la técnica de mapeo objeto-relacional tenemos, por un lado, una base de datos con su estructura relacional y, por otro lado, clases Java que imitan la estructura de la base de datos.

Estas clases Java se denominan **clases de entidad** y deben cumplir unas normas:

- Serán *JavaBeans*, es decir, cumplirán con las mismas normas que deben cumplir éstos: tener capacidad de persistencia, contar con un constructor vacío, y atributos privados y accesibles mediante métodos *get/set*.
- Utilizarán anotaciones para indicar información adicional a cada atributo o clase, como si se corresponde con una clave principal, con una clave ajena, etc. Las anotaciones van precedidas del símbolo @.

¿Cómo sería la implementación de la clase de entidad *Cliente*?

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.List;

@Entity
@NamedQuery(name="Cliente.findAll", query="SELECT c FROM Cliente c")
public class Cliente implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    private String nif;

    private String ciudad;

    private String domicilio;

    private String nombre;

    private String tlf;

    //bi-directional many-to-one association to Factura
    @OneToMany(mappedBy="cliente")
    private List<Factura> facturas;

    public Cliente() {
    }

    public String getNif() {
        return this.nif;
    }
}
```

```
public void setNif(String nif) {
    this.nif = nif;
}

public String getCiudad() {
    return this.ciudad;
}

public void setCiudad(String ciudad) {
    this.ciudad = ciudad;
}

public String getDomicilio() {
    return this.domicilio;
}

public void setDomicilio(String domicilio) {
    this.domicilio = domicilio;
}

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

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

public String getTlf() {
    return this.tlf;
}

public void setTlf(String tlf) {
    this.tlf = tlf;
}

public List<Factura> getFacturas() {
    return this.facturas;
}

public void setFacturas(List<Factura> facturas) {
    this.facturas = facturas;
}

public Factura addFactura(Factura factura) {
    getFacturas().add(factura);
    factura.setCliente(this);

    return factura;
}

public Factura removeFactura(Factura factura) {
    getFacturas().remove(factura);
    factura.setCliente(null);

    return factura;
}

}
```

Observa cómo se utilizan las siguientes anotaciones:

- `@Entity` delante de la cabecera de la clase para indicar que se trata de una clase de entidad.
- `@Id` delante del `nif` para indicar que se trata de la clave principal.
- `@OneToMany(mappedBy="cliente")` delante del atributo `facturas` para indicar que dicho atributo servirá para representar la asociación *uno a muchos* entre el cliente y la factura.

Aprenderemos cómo crear estas clases de entidad y comprenderemos mejor el por qué de cada una de las anotaciones que contienen.

Las anotaciones

Ya sabemos que las clases de entidad utilizan anotaciones. Pero, ¿qué son realmente las anotaciones?

Las anotaciones Java proveen de un sistema para añadir metadatos al código fuente que estarán disponibles para la aplicación en tiempo de ejecución.

¿Qué son los *metadatos*?

La traducción literal de metadato es *sobre dato* y hace referencia a un dato que describe a otro dato. En aplicaciones empresariales se utiliza para añadir información de configuración a un objeto cuya misión es servir como recurso a un proyecto.

En la clase de entidad del apartado anterior, la anotación `@Id` servía como dato para indicar que el dato `dni` es una clave principal.

Obtén más información en la wikipedia. <https://es.wikipedia.org/wiki/Metadatos>

Un ejemplo de anotación que con seguridad te has encontrado en más de una ocasión es la anotación `@Override` delante de un método, para indicar que dicho método viene heredado de una clase base y está siendo sobrescrito.

Ejercicio práctico

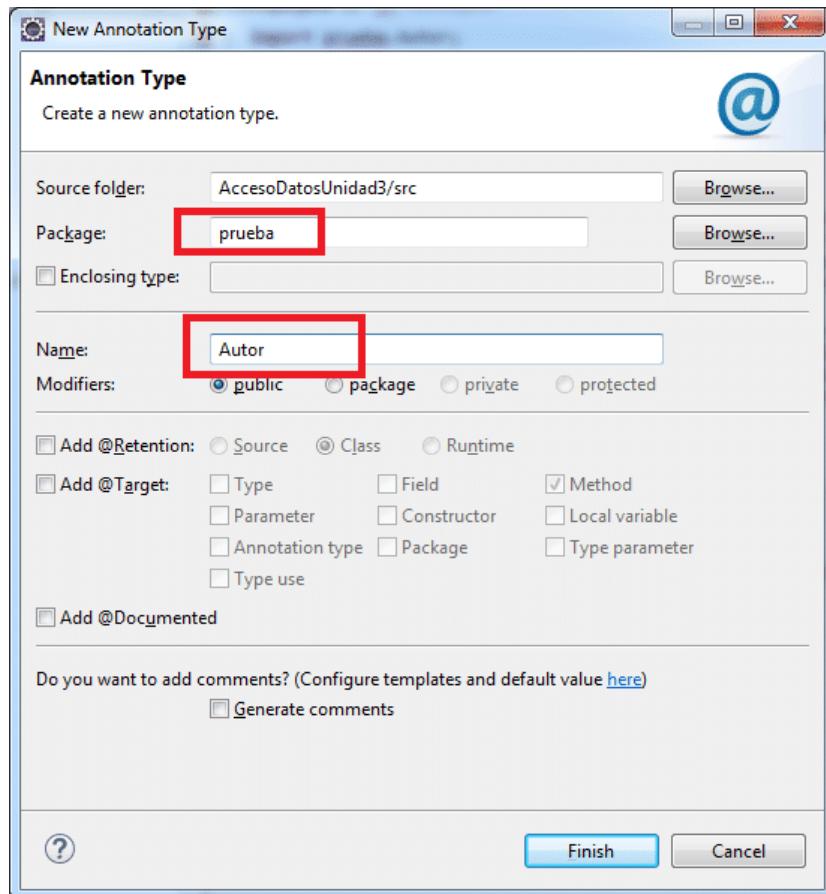
Para que comprendas mejor lo que son las anotaciones Vas a crear tu propia anotación personalizada y a utilizarla después.

Seguiremos los siguientes pasos:

- Crearemos la anotación personalizada `@Autor`, que servirá para añadir información a las clases y métodos que queramos sobre quién es el autor que desarrolló dichas clases o métodos. La anotación contará con las propiedades *nombre* y *direccion*.
- Crearemos una clase llamada `Coche` y la anotaremos. Dentro de la clase `Coche` implementaremos un método llamado `acelerar()`, que también anotaremos.
- En la clase `Principal` crearemos un objeto de la clase `Coche` y accederemos a los datos suministrados por la anotación `@Autor`.

1. Creamos la anotación personalizada `@Autor`

- Crea un proyecto nuevo denominado *pruebaAnotaciones* (*File / New / Java Project*).
- Haz clic derecho sobre el nombre del nuevo proyecto y selecciona en el menú contextual *New / Annotation*.



En el cuadro de diálogo *New Annotation Type* rellena los datos como ves en la imagen para crear la anotación `Autor` en el paquete `prueba`. Termina haciendo clic en el botón *Finish*.

```
package prueba;  
  
public @interface Autor {  
}
```

Por ahora tu anotación tiene este aspecto. Las anotaciones son como interfaces especiales. Observa que la palabra interface va precedida por el símbolo de arroba.

Ahora, completa el código de la anotación de la siguiente manera:

```
package prueba;  
  
import java.lang.annotation.*;  
  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Autor {  
    String nombre() default "Telefónica";  
    String direccion() default "Distrito Telefónica";  
}
```

Ahora la anotación cuenta con los atributos o datos *nombre* y *direccion* cuyos valores predeterminados son "*Teléfonica*" y "*Distrito Telefónica*".

La anotación `@Retention(RetentionPolicy.RUNTIME)` permite que la anotación `@Autor` esté disponible en tiempo de ejecución.

2. Creamos una clase llamada *Coche* y la anotamos

La nueva clase *Coche* tendrá el siguiente código:

```
import prueba.Autor;  
  
@Autor  
public class Coche {  
    private String marca;  
    private String modelo;  
    private int velocidad;  
  
    public Coche(String marca, String modelo) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.velocidad = 0;  
    }  
  
    public String getMarca() {  
        return marca;  
    }
```

```
public String getModelo() {
    return modelo;
}
public int getVelocidad() {
    return velocidad;
}

@Autor(nombre="Perico de los palotes", direccion="C/ Palotes, 54")
public void acelerar() {
    this.velocidad = this.velocidad + 10;
}

@Override
public String toString() {
    return "Coche [marca=" + marca + ", modelo=" + modelo +",
velocidad=" + velocidad + "]";
}

}
```

Hemos anotado la nueva clase *Coche* sin especificar ningún valor para los atributos *nombre* y *dirección*, por lo tanto, habrá tomado los valores predeterminados "*Telefónica*" y "*Distrito Telefónica*". Si embargo, el método *acelerar()* ha sido anotado especificando el valor "*Perico de los Palotes*" para el atributo *nombre* y "*C/ Palotes, 54*" para el atributo *direccion*.

3. En la clase *Principal*/creamos un objeto de la clase *Coche* y accedemos a los datos suministrados por la anotación *@Autor*.

Por último, crea la clase *Principal* con el siguiente código:

```
import prueba.Autor;

public class Principal {

    public static void main(String[] args) throws NoSuchMethodException,
SecurityException {
        Coche miCoche = new Coche("Ford", "Fiesta");
        System.out.println(miCoche);

        // Accediendo a los datos de la anotación del método.
        Autor a1 =
miCoche.getClass().getMethod("acelerar").getAnnotation(Autor.class);
        System.out.println("Nombre autor: " + a1.nombre());
        System.out.println("Dirección autor: " + a1.direccion());

        // Accediendo a los datos de la anotación de la clase.
        Autor a2 = miCoche.getClass().getAnnotation(Autor.class);
        System.out.println("Nombre autor: " + a2.nombre());
        System.out.println("Dirección autor: " + a2.direccion());

    }

}
```

Veamos las partes más importantes de la clase *Principal*:

Coche miCoche = new Coche("Ford", "Fiesta");

Creamos un objeto de la clase *Coche* y después mostramos su estado invocando al método *toString()* con esta línea:

System.out.println(miCoche);

Recuerda que *toString()* es el método al que se invoca por defecto cuando no se especifica el método a ejecutar.

**Autor a1 =
miCoche.getClass().getMethod("acelerar").getAnnotation(Autor.class);**

La expresión *miCoche.getClass()* devuelve un objeto de tipo *Class* que provee información sobre la clase a la que pertenece el objeto. De esta forma, estamos también obteniendo información sobre el método *acelerar()* usando la expresión *getMethod("acelerar")*. Concretamente, estamos accediendo a los datos de la anotación.

Autor a2 = miCoche.getClass().getAnnotation(Autor.class);

Aquí estamos accediendo a los datos de la anotación de la clase.

¿Qué son los frameworks ORM?

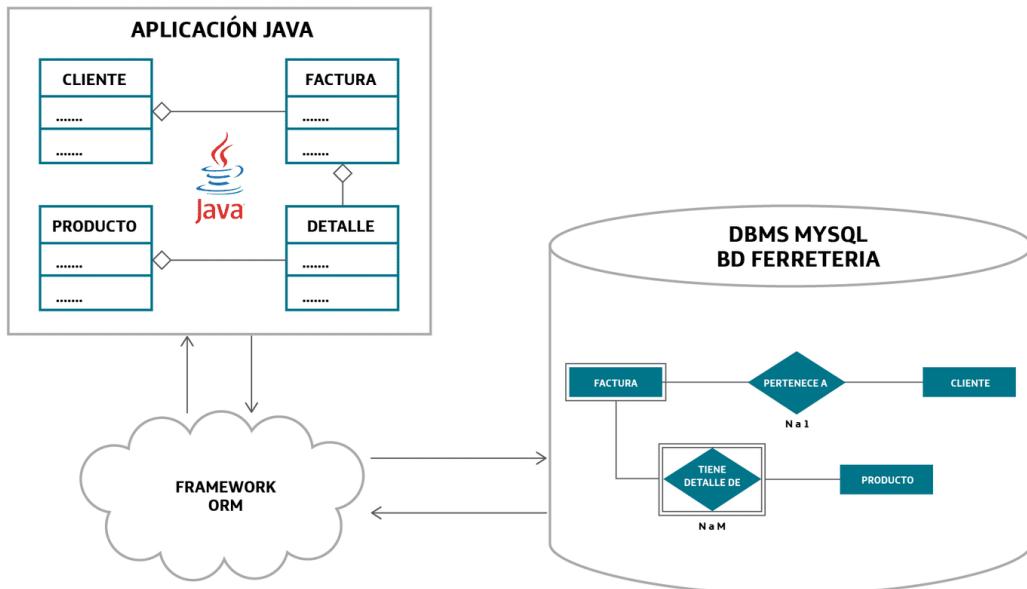
Framework significa *marco de trabajo*, y consiste en un conjunto de prácticas y normas que ayudan a solucionar un tipo de problema estandarizado.

Los *frameworks* en el contexto de la ingeniería de software ofrecen, entre otras cosas, una **librería o API que da soporte completo para el desarrollo de un determinado tipo de aplicaciones**.

Pues bien, un **Framework ORM** es un marco de trabajo que ofrece un soporte completo para el desarrollo de aplicaciones con bases de datos relacionales y se basa en el procedimiento de Mapeo Objeto-Relacional.

Las siglas ORM vienen del inglés *Object Relational Mapping*

El Framework ORM se sitúa en una capa intermedia entre las bases de datos y nuestra aplicación Java. Es el encargado de convertir las tablas físicas de la base de datos relacional en objetos Java que formarán una base de datos orientada a objetos virtual. La aplicación Java realizará las actualizaciones necesarias sobre los objetos Java y de nuevo será el Framework ORM el encargado de volcar dichos cambios en la base de datos física.



JPA (Java Persistence API)

Forma parte del estándar Java EE y está definida en el paquete `javax.persistence`. No se trata de un API completo, sino más bien de una especificación que requiere de una implementación. Varios productos comerciales implementan JPA; uno de ellos, el que vamos a utilizar en este curso, es *EclipseLink*.

Hibernate

Otro Framework ORM bastante extendido en la comunidad de desarrolladores Java, aunque también está disponible para la plataforma .NET. Fue desarrollado por la organización JBoss.

MyBatis

Otra herramienta ORM a cargo de Apache Tomcat Foundation.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- El **mapeo objeto-relacional** es un proceso a través del cual se crea, a partir de una base de datos relacional, un modelo de objetos que simula una base de datos orientada a objetos virtual. De esta manera, nuestra aplicación trabajará contra objetos Java en lugar de hacerlo directamente con la base de datos.
- Los **Frameworks ORM (*Object-Relational Mapping*)** son los encargados de llevar a cabo el mapeo objeto-relacional.
- Los Frameworks ORM utilizan **clases de entidad que cumplen la especificación de los JavaBeans y emplean anotaciones.**

3.2. Creación de un proyecto JPA en Eclipse contra una BD MySQL.



Índice

Introducción.....	3
Objetivos.....	3
Configuración del proyecto JPA.....	4
La perspectiva Database Development de Eclipse	4
Introducción a JPA	4
Configurar un proyecto para trabajar con JPA	5
La estructura del nuevo proyecto	7
El fichero persistence.xml	9
Aplicación práctica.....	13
Inventario de la FERRETERIA	13
Clases Java.....	17
Facturas con sus detalles	18
Nuevas anotaciones	22
Las clases de entidad Detalle y DetallePK.....	23
Buscar información de un cliente	27
Añadir un cliente.....	30
Las operaciones CRUD con JPA	31
Java Persistence Query Language (JPQL)	32
Listado de productos para reponer	36
Añadir una factura	36
Despedida	42
Resumen.....	42

Introducción

Objetivos

En esta unidad perseguimos los siguientes objetivos:

- Desarrollar aplicaciones Java que accedan a bases de datos relacionales, apoyándose en algún *Framework* ORM.
- Configurar un proyecto Eclipse para trabajar con el *Framework* ORM JPA.
- Interactuar con el *Framework* JPA dentro del proyecto, utilizando las clases y métodos que JPA pone a disposición de sus usuarios.

Configuración del proyecto JPA

La perspectiva Database Development de Eclipse

Eclipse tiene una perspectiva denominada *Database Development* que te permitirá configurar una conexión con una base de datos.

Aunque no es un paso absolutamente necesario, resulta muy útil porque te permitirá inspeccionar la estructura de la base de datos directamente desde Eclipse.

Además, cuando llegue el momento de crear la aplicación JPA, Eclipse se basará en la información suministrada por esta conexión para crear automáticamente por nosotros las clases de entidad, ahorrándonos ese trabajo.

El siguiente video muestra los pasos para cambiar a la perspectiva *Database Development* y añadir una conexión para inspeccionar la base de datos *FERRETERIA*.
<https://vimeo.com/telefonicaed/review/271423998/50991e5c3b>

Introducción a JPA

Ya sabemos que un *Framework ORM* implementa la técnica de Mapeo objeto-relacional, proporcionando una estructura de objetos que representa la base de datos.

Pues bien, se denomina **capa de persistencia** a la capa que encapsula el comportamiento para mantener dichos objetos.

JPA (Java Persistence API) es el *framework* estándar proporcionado por Java Enterprise Edition (Java EE) para la capa de persistencia que implementa el concepto de ORM.

Características de JPA (*Java Persistence API*):

- **Persistencia utilizando POJOs** (https://es.wikipedia.org/wiki/Plain_Old_Java_Object): cualquier clase Java podemos convertirla en una clase de entidad, simplemente agregando anotaciones.
- **No intrusivo:** JPA es una capa separada de los objetos a persistir. Las clases de entidad no requieren de ninguna funcionalidad en particular, ni saber de la existencia del API JPA.
- **Consultas utilizando objetos java:** JPA permite ejecutar consultas expresadas en objetos Java y sus relaciones sin necesidad de utilizar el lenguaje SQL. Las consultas son traducidas por el API JPA en el código SQL equivalente.

- **Configuración simplificada:** muchas de las opciones de configuración de JPA tienen una valor por defecto. Sin embargo, si queremos personalizarlas, es muy fácil hacerlo, ya sea con anotaciones o por medio de un archivo XML de configuración.
- **Integración con la especificación Java EE:** JPA se sitúa, dentro de las aplicaciones Java, en una capa independiente del resto, por lo que se integra perfectamente con el resto de capas de la aplicación sin alterarlas.

JPA **no es una herramienta completa, sino que puede ser considerada como una especificación que necesita un proveedor que la implemente.** En este sentido, Eclipse nos asiste a la hora de configurar un proyecto JPA para descargar de internet una implementación.

Son implementaciones de JPA: EclipseLink, Apache OpenJPA, Toplink, Hibernate, etc.

En el siguiente apartado crearás un proyecto JPA y podrás comprobar cómo Eclipse te permite descargar una implementación.

Configurar un proyecto para trabajar con JPA

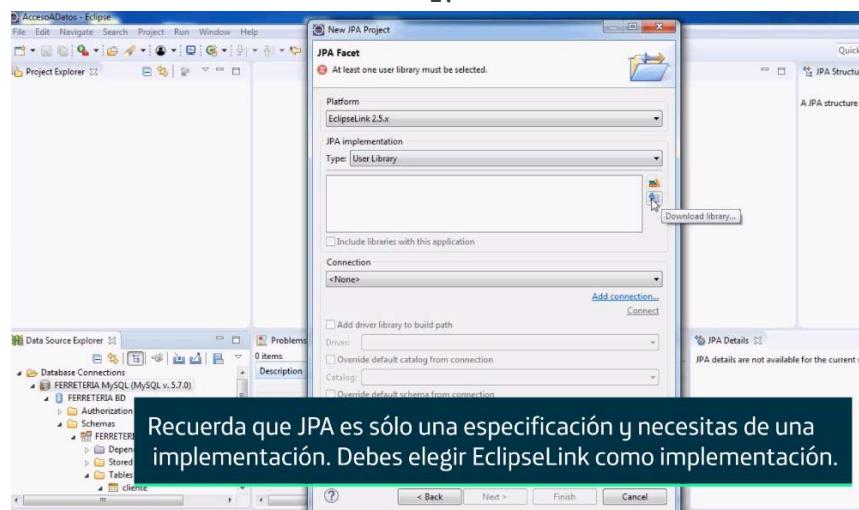
En el siguiente apartado, prepararás un proyecto Java perfectamente configurado para trabajar con JPA.

Eclipse cuenta con la perspectiva JPA para este tipo de aplicaciones. Si sigues los pasos que verás en el vídeo, podrás crear un proyecto de tipo *JPA Project* y crear las clases de entidad automáticamente. Lo harás con ayuda de la información suministrada por la conexión a la base de datos *FERRETERIA* que dejaste preparada en el anterior vídeo.

<https://vimeo.com/telefonicaed/review/271424014/d16ab5ff4b>

Vamos a recordar **tres pasos importantes** que has llevado a cabo **para configurar tu proyecto JPA:**

1.



Elección de EclipseLink.

Como comentamos anteriormente, JPA necesita de un fabricante que implemente su especificación. En el paso que ves en la imagen hemos escogido *EclipseLink* como implementación, pero no tenemos su librería descargada en nuestro equipo, por lo tanto, hemos pulsado el botón *Download Library* para descargarlo de internet. El software descargado quedará situado dentro de una carpeta del espacio de trabajo; puedes utilizar el explorador de Windows para comprobarlo.

El botón situado justo encima de *Download Library*, representado por una pila de libros, es el botón *Manage Libraries* y se utiliza para crear una nueva librería o editar las existentes. Será útil en caso de que tengamos el software ya descargado en alguna ubicación de nuestro sistema de archivos. Pero, para nuestro ejemplo, no lo hemos utilizado.

2.



Proceso de descarga de EclipseLink.

Una vez terminado el proceso de descarga, la nueva librería aparecerá dentro del apartado *JPA Implementation*. En los siguientes proyectos JPA que crees dentro del mismo espacio de trabajo ya aparecerá la librería y no será necesario repetir el paso anterior: sólo dejaremos marcada o desmarcada la casilla de verificación, en función de si deseamos utilizarla o no.

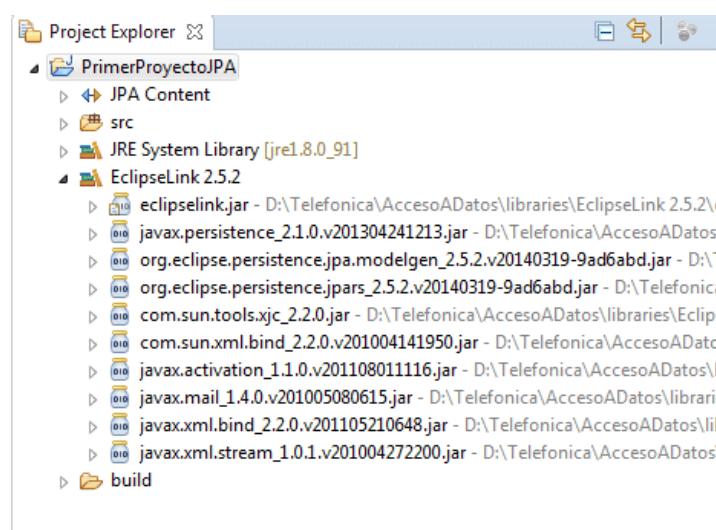
3.



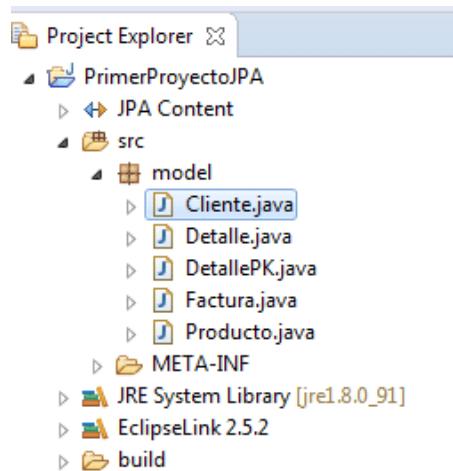
Recuerda que pudiste crear automáticamente las clases de entidad gracias a que antes dejaste abierta en Eclipse una conexión a la base de datos *FERRETERIA*, que fue utilizada por el sistema como guía para desarrollar el código de manera automática.

La estructura del nuevo proyecto

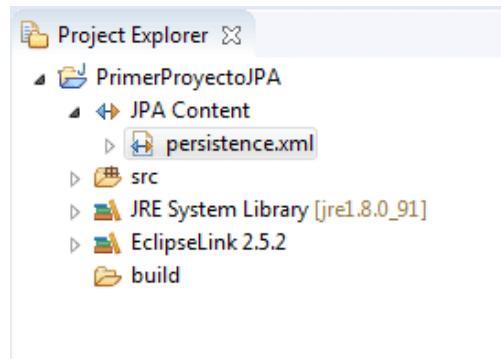
Estructura del proyecto JPA:



Nuestro proyecto no es más que un proyecto de consola que tiene agregadas las librerías y características de un proyecto JPA. En primer lugar, además de las librerías estándar del JRE, cuenta con las **librerías de EclipseLink**. Además, puedes ver junto a cada archivo JAR la ubicación dentro del sistema de archivos en la que está situado.



Las clases de entidad quedan situadas en un paquete denominado ***model***.



Los proyectos JPA cuentan con un importante archivo de configuración denominado ***persistence.xml*** que estudiaremos más detenidamente en el siguiente apartado.

IMPORTANTE: Tu nuevo proyecto está pensado para trabajar con una base de datos MySQL, así que no olvides importar el driver de MySQL en el proyecto como aprendiste en la unidad anterior (clic derecho en el nombre del proyecto / Properties / Java Build Path / Add External jars).

El fichero persistence.xml

El archivo **persistence.xml** contiene la configuración de las denominadas **Unidades de persistencia**, un concepto de vital importancia y que aclararemos en este apartado.

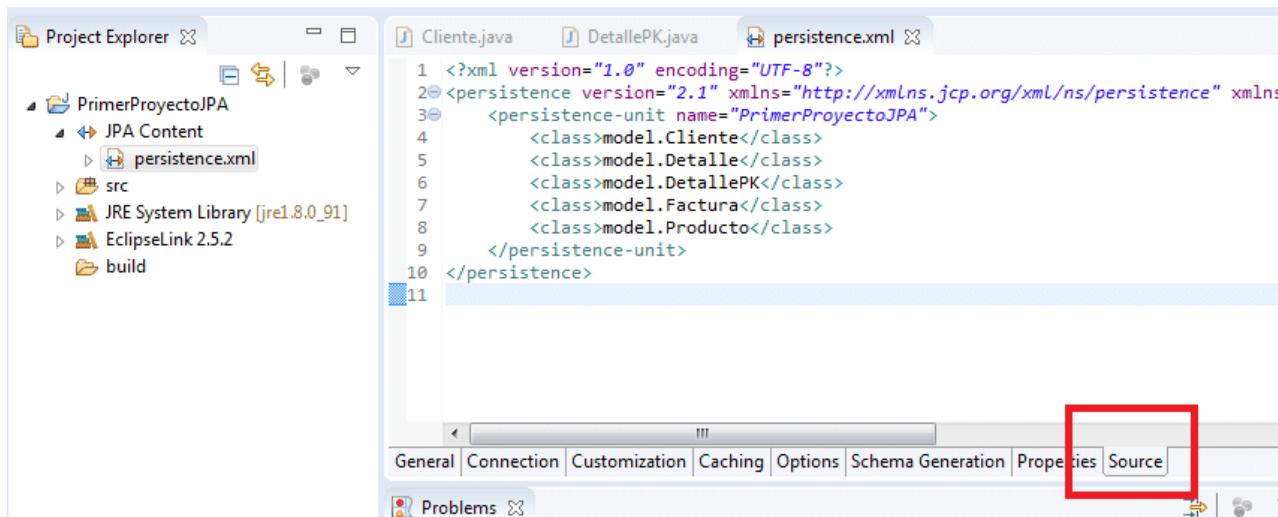
Cada **unidad de persistencia** encierra la configuración de conexión con una base de datos determinada.

Una unidad de persistencia va encerrada entre las etiquetas:

```
<persistence-unit name="nombreUnidadPersistencia">  
....  
</persistence-unit>
```

Donde *nombreUnidadPersistencia* es un identificador de vital importancia para hacer referencia a una conexión u otra dentro del código de nuestra aplicación.

Si una misma aplicación JPA debe acceder a varias bases de datos, tendrá que existir una unidad de persistencia para cada una de ellas.

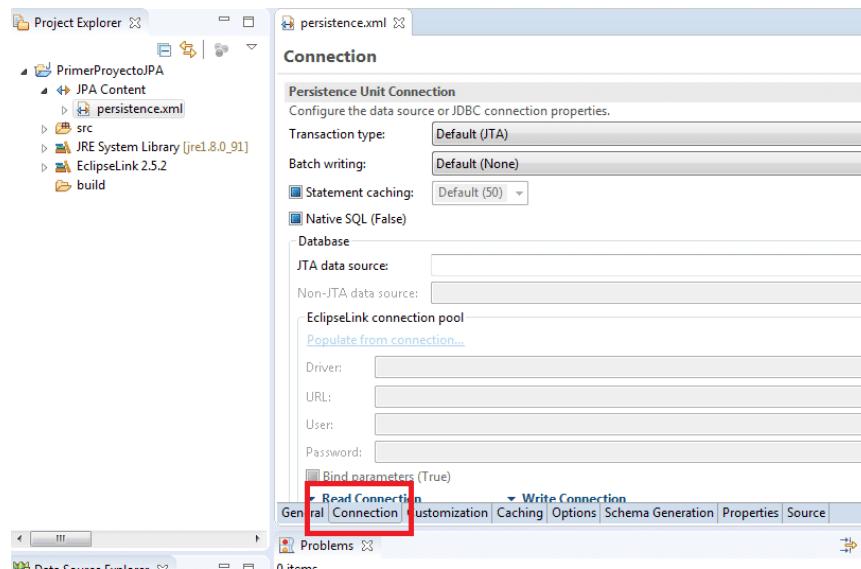


Utiliza el explorador de proyectos para abrir el archivo *persistence.xml* y abre la pestaña *Source* para ver la vista de código XML. Observa que, por defecto, Eclipse ha dado nombre a la unidad de persistencia utilizando el nombre del proyecto. Puedes dejarlo así o cambiarlo, si lo deseas.

Pero todavía hay algo importante que falta por añadir a la unidad de persistencia: los datos de conexión con la base de datos *FERRETERIA*, el equivalente a la cadena de conexión que utilizábamos en JDBC.

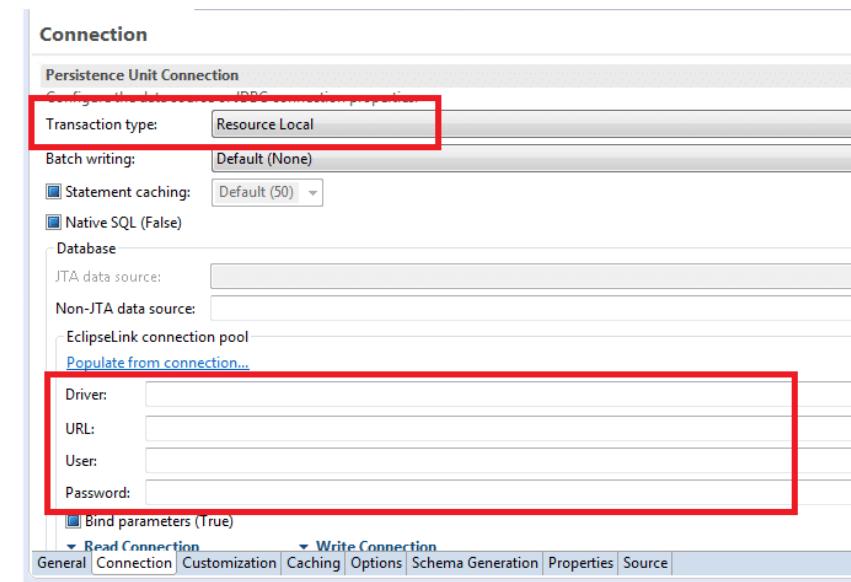
Podríamos llevar a cabo esta tarea a mano, pero **vamos a volver a utilizar los recursos que nos presta Eclipse para ahorrarnos trabajo**. Realiza los pasos que te indicamos a continuación:

1.



Continúa manteniendo abierto el archivo *persistence.xml*, pero esta vez en la ficha *Connection*.

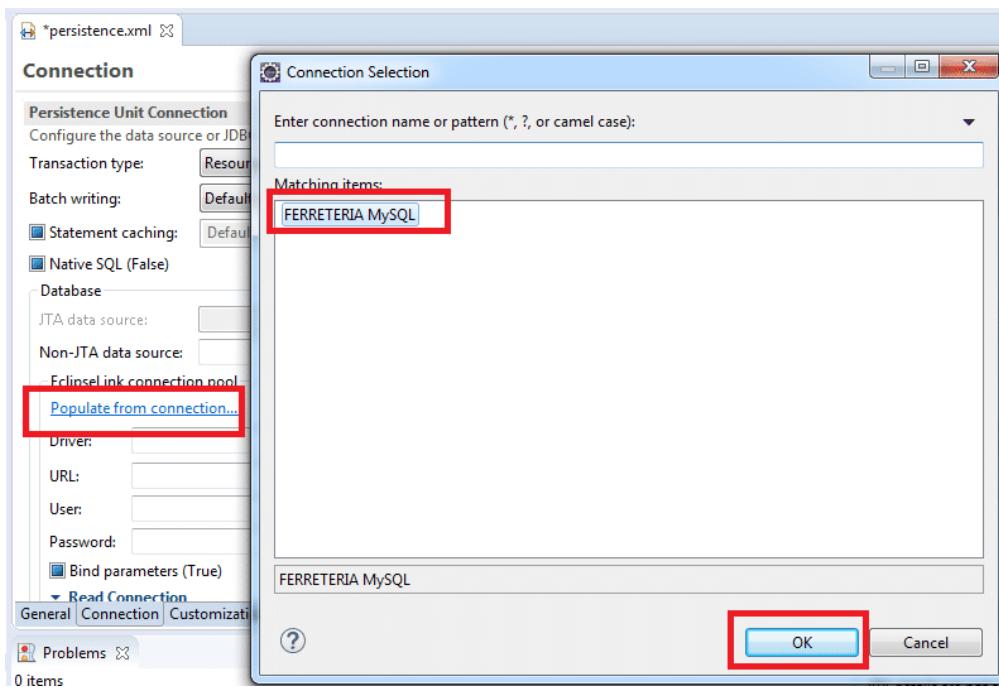
2.



Selecciona *Resource Local* en la lista desplegable *Transaction type* y verás cómo se habilitan las cajas de texto para especificar los valores de *Driver*, *URL*, *User* y *Password*.

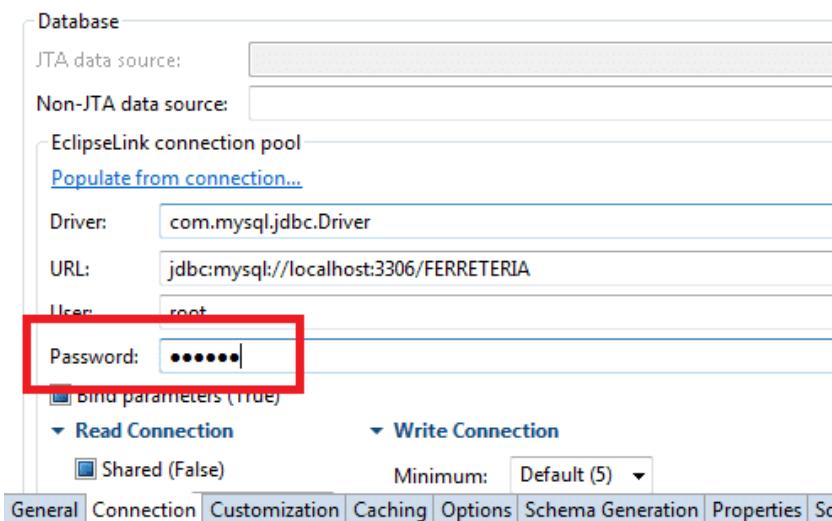
Eclipse utilizará la información de la perspectiva *Database Development* para localizar las conexiones a bases de datos abiertas.

3.



Haz clic en el enlace *Populate from connection*. Selecciona la conexión *FERRETERIA MySQL* y pulsa *OK*.

4.



Eclipse ha completado todos los datos de conexión menos el *Password*, que debes rellenarlo tú con la contraseña que especificaste cuando instalaste MySQL.

El archivo persistence.xml ya ha quedado perfectamente configurado para comenzar a trabajar. Éste es su contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="PrimerProyectoJPA" transaction-
  type="RESOURCE_LOCAL">
    <class>model.Cliente</class>
    <class>model.Detalle</class>
    <class>model.DetallePK</class>
    <class>model.Factura</class>
    <class>model.Producto</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/FERRETERIA"/>
      <property name="javax.persistence.jdbc.user"
        value="root"/>
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password"
        value="amelia"/>
    </properties>
  </persistence-unit>
</persistence>
```

Hasta ahora, todo lo que has hecho han sido labores de configuración del proyecto. En los siguientes apartados escribirás el código de tus aplicaciones JPA.

Aplicación práctica

Inventario de la FERRETERIA

Ahora es el momento de escribir el código necesario para lograr que nuestra aplicación sea capaz de mostrar al usuario el inventario de *FERRETERIA*, es decir, el listado de artículos disponibles.

Lo que realmente hará nuestra aplicación es obtener una colección de objetos de la clase *Producto*, uno por cada fila existente en la tabla *PRODUCTO* de la base de datos.

En esta tarea entrará en juego la clase de entidad *Producto*, por lo que, antes de nada, vamos a revisarla más exhaustivamente.

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.List;

@Entity
@NamedQuery(name="Producto.findAll", query="SELECT p FROM Producto p")
public class Producto implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    private String codigo;

    private String descripcion;

    private float minimo;

    private float precio;

    private float stock;

    //bi-directional many-to-one association to Detalle
    @OneToMany(mappedBy="producto")
    private List<Detalle> detalles;

    public Producto() {
    }

    public String getCodigo() {
        return this.codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }
}
```

```
public String getDescripcion() {
    return this.descripcion;
}

public void setDescripcion(String descripcion) {
    this.descripcion = descripcion;
}

public float getMinimo() {
    return this.minimo;
}

public void setMinimo(float minimo) {
    this.minimo = minimo;
}

public float getPrecio() {
    return this.precio;
}

public void setPrecio(float precio) {
    this.precio = precio;
}

public float getStock() {
    return this.stock;
}

public void setStock(float stock) {
    this.stock = stock;
}

public List<Detalle> getDetalles() {
    return this.detalles;
}

public void setDetalles(List<Detalle> detalles) {
    this.detalles = detalles;
}

public Detalle addDetalle(Detalle detalle) {
    getDetalles().add(detalle);
    detalle.setProducto(this);

    return detalle;
}

public Detalle removeDetalle(Detalle detalle) {
    getDetalles().remove(detalle);
    detalle.setProducto(null);

    return detalle;
}

}
```

Analicemos paso a paso la clase de entidad *Producto* y sus anotaciones:

@Entity

Lo más importante es la anotación `@Entity` que indica que se trata de una clase de entidad. Esta anotación va a menudo acompañada de la anotación `@Table`, que indica el nombre de la tabla física en la base de datos que se relaciona con la clase de entidad.

```
@Entity
@Table(name="producto")
public class Producto implements Serializable {
    ..
}
```

Cuando el nombre de la tabla física coincide con el nombre de la clase de entidad no es necesario utilizar la anotación `@Table`.

Por convención, las clases de entidad deberían ser nombradas con un sustantivo en singular. Ejemplo: para una tabla física llamada *Alumnos* en la base de datos, la clase de entidad debería llamarse *Alumno*.

@NamedQuery(name="Producto.findAll", query="SELECT p FROM Producto p")

Esta anotación define lo que se llama una **consulta con nombre**. Bajo el identificador `Producto.findAll` podemos hacer referencia a la consulta `SELECT p FROM Producto p`.

La consulta está definida en el lenguaje JPQL (*Java Persistence Query Language*), lenguaje de consulta orientado a objetos que forma parte de la especificación JPA y que estudiaremos más detenidamente en otro apartado.

@Id

Esta anotación se coloca delante de un atributo para indicar que se trata de la clave principal. En ocasiones va acompañada de la anotación `@GeneratedValue` para especificar la forma en la que la clave principal será generada.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int idCliente;
```

En este ejemplo estamos indicando que el atributo *idCliente* es la clave principal y que queremos que sea de tipo identidad, es decir, que se vaya auto generando de manera incremental.

@OneToOne(mappedBy="producto")

Esta anotación indica que existe una asociación de tipo *uno a muchos* entre la tabla producto y la tabla detalle.

```
//bi-directional many-to-one association to Detalle  
@OneToOne(mappedBy="producto")  
private List<Detalle> detalles;
```

El atributo *detalles* es una colección de elementos de tipo *Detalle* que contendrá para cada objeto *Producto* tantos objetos *Detalle* como ventas existan para dicho producto.

Este tipo de anotaciones pueden ser:

```
@OneToOne  
@ManyToOne  
@OneToOne
```

public Detalle addDetalle(Detalle detalle) {...}

Puesto que cada objeto *Producto* puede estar asociado a varias ventas (objetos *Detalle*), este método permite añadir un nuevo objeto *Detalle*.

public Detalle removeDetalle(Detalle detalle) {...}

Puesto que cada objeto *Producto* puede estar asociado a varias ventas (objetos *Detalle*), este método permite eliminar una venta (*Detalle*).

Llegó el momento de crear la clase *Principal* y ejecutar el proyecto.

¡Ojo! No la crees en el paquete *model*, déjala fuera, ya que el paquete *model* está pensado para contener sólo las clases de entidad.

```
import java.util.List;  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Persistence;  
import javax.persistence.TypedQuery;
```

```
import model.Producto;  
  
public class Principal {  
  
    public static void main(String[] args) {  
  
        listado();  
  
    }  
  
    public static void listado() {  
        EntityManagerFactory factoria =  
Persistence.createEntityManagerFactory("PrimerProyectoJPA");  
        EntityManager em = factoria.createEntityManager();  
        TypedQuery<Producto> query =  
em.createNamedQuery("Producto.findAll", Producto.class);  
        List<Producto> productos = query.getResultList();  
  
        for (Producto p : productos) {  
            System.out.println(p.getCodigo() + " - " +  
p.getDescripcion() +  
                    " - " + p.getPrecio() + " € - " + p.getStock() + "  
unidades.");  
        }  
    }  
}
```

Si ya has ejecutado el programa, habrás obtenido el listado de artículos del almacén. Ahora estudiaremos las nuevas clases Java que hemos empleado.

Clases Java

El paquete *javax.persistence*:

Las clases que forman parte de la librería que implementa JPA (*EclipseLink*, en nuestro caso) se encuentran situadas en el paquete *javax.persistence*. Vamos a estudiar las clases que hemos utilizado en el ejemplo anterior.

javax.persistence.Persistence

Utilizamos la clase *Persistence* para invocar a su método estático *createEntityManagerFactory()* pasando como argumento el nombre de la unidad de persistencia. Esto nos permitirá obtener un objeto de la clase *EntityManagerFactory*, imprescindible para interactuar con la base de datos a través de la unidad de persistencia.

javax.persistence.EntityManagerFactory

Necesitamos nuestro objeto *EntityManagerFactory* para obtener el *EntityManager* a través de la llamada al método *createEntityManager()*.

javax.persistence.EntityManager

El objeto *EntityManager* actúa como administrador de las clases de entidad, permitiéndonos efectuar todas las operaciones CRUD (*Create, Read, Update and Delete*) en la base de datos a través de las clases de entidad.

javax.persistence.TypedQuery

TypeQuery es una Interfaz utilizada para controlar la ejecución de consultas tipificadas, es decir, podemos especificar el tipo de objetos que devolverá la consulta.

Sobre el objeto *TypeQuery* se podrá invocar al método *getResultSet()*, que retornará el conjunto de objetos resultado de la ejecución de la consulta; para nuestro ejemplo, todos los productos de la *FERRETERIA*.

```
List<Producto> productos = query.getResultSet();
```

Facturas con sus detalles

Vamos a completar el programa anterior para que, además de mostrar el inventario, muestre un listado de facturas, incluyendo debajo de cada una sus líneas de detalle.

Ten en cuenta que cada objeto de la clase *Factura* es un compuesto que lleva, además de los datos propios de la factura, el objeto *Cliente* asociado y la colección de detalles (objetos *Detalle*). Y que por cada objeto *Detalle* podemos, además, acceder a los datos del objeto *Producto* relacionado.

En conclusión, **obteniendo la relación de facturas obtenemos el resto de la información asociada a través de todo el modelo de objetos**. El programa modificado quedaría así:

```
import java.util.List;

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

import model.Detalle;
import model.Factura;
import model.Producto;

public class Principal {

    public static void main(String[] args) {
        inventario();
    }
}
```

```
        System.out.println();
        facturasDetalles();
    }

    public static void inventario() {
        EntityManagerFactory factoria =
Persistence.createEntityManagerFactory("PrimerProyectoJPA");
        EntityManager em = factoria.createEntityManager();
        TypedQuery<Producto> query =
em.createNamedQuery("Producto.findAll", Producto.class);
        List<Producto> productos = query.getResultList();

        for (Producto p : productos) {
            System.out.println(p.getCodigo() + " - " +
p.getDescripcion() + " - " + p.getPrecio() + " € - " + p.getStock() + " "
unidades.);
        }
    }

    public static void facturasDetalles() {
        EntityManagerFactory factoria =
Persistence.createEntityManagerFactory("PrimerProyectoJPA");
        EntityManager em = factoria.createEntityManager();
        TypedQuery<Factura> query =
em.createNamedQuery("Factura.findAll", Factura.class);
        List<Factura> facturas = query.getResultList();

        for (Factura f : facturas) {
            System.out.println(f.getNumero() + " - " + f.getFecha() +
" - " + f.getCliente().getNombre());
            for (Detalle d : f.getDetalles()) {
                Producto p = d.getProducto();
                System.out.println("      " + p.getCodigo() + " - " +
+ p.getDescripcion() + " - " + d.getPrecio() + "€ " + d.getUnidades());
            }
        }
    }
}
```

Para cada factura estamos recorriendo sus líneas de detalle a partir del método *getDetalles()*. De la misma manera, para cada detalle accedemos al objeto *Producto* asociado y lo guardamos en un objeto cuya referencia se encuentra en la variable *p*.

Si has ejecutado el programa, es muy posible que arroje algún error y estés viendo la traza del error en la consola de Eclipse.

IMPORTANTE: La herramienta de generación automática de clases de entidad que nos brinda Eclipse es muy útil, pero puede cometer pequeños fallos a la hora de establecer los tipos de datos de las propiedades. En este caso, el programa no funciona y provoca un error de ejecución. Al examinar la traza del error podemos encontrar el siguiente texto:

Exception Description: The object [true], of class [class java.lang.Boolean], from mapping [org.eclipse.persistence.mappings.DirectToFieldMapping[pagado-->FACTURA.PAGADO]] with descriptor [RelationalDescriptor(model.Factura --> [DatabaseTable(FACTURA)])], could not be converted to [class java.lang.Byte].

El conflicto lo está provocando la clase *Factura* porque el atributo *pagado* está declarado como tipo *byte* y, sin embargo, el campo *PAGADO* de la base de datos fue declarado como tipo *BOOL* (aunque internamente sea un número entero de un solo dígito).

Solucionamos el problema declarando el atributo *pagado* en la clase de entidad como tipo *boolean*.

Así es como queda el código Java de la entidad *Factura* tras cambiar el tipo de dato del atributo *pagado* y sus métodos *get/set* asociados.

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;
import java.util.List;

@Entity
@NamedQuery(name="Factura.findAll", query="SELECT f FROM Factura f")
public class Factura implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    private int numero;

    @Temporal(TemporalType.DATE)
    private Date fecha;

    private boolean pagado;

    //bi-directional many-to-one association to Detalle
    @OneToMany(mappedBy="factura")
    private List<Detalle> detalles;

    //bi-directional many-to-one association to Cliente
    @ManyToOne
    @JoinColumn(name="NIF")
    private Cliente cliente;

    public Factura() {
    }

    public int getNumero() {
        return this.numero;
    }

    public void setNumero(int numero) {
        this.numero = numero;
    }
}
```

```
public Date getFecha() {
    return this.fecha;
}

public void setFecha(Date fecha) {
    this.fecha = fecha;
}

public boolean getPagado() {
    return this.pagado;
}

public void setPagado(boolean pagado) {
    this.pagado = pagado;
}

public List<Detalle> getDetalles() {
    return this.detalles;
}

public void setDetalles(List<Detalle> detalles) {
    this.detalles = detalles;
}

public Detalle addDetalle(Detalle detalle) {
    getDetalles().add(detalle);
    detalle.setFactura(this);

    return detalle;
}

public Detalle removeDetalle(Detalle detalle) {
    getDetalles().remove(detalle);
    detalle.setFactura(null);

    return detalle;
}

public Cliente getCliente() {
    return this.cliente;
}

public void setCliente(Cliente cliente) {
    this.cliente = cliente;
}

}
```

Nuevas anotaciones

Estas son algunas anotaciones nuevas que han aparecido en la clase de entidad *Factura*:

@Temporal(TemporalType.DATE)

Esta anotación está colocada delante del atributo *fecha* para indicar cómo debe ser convertido el campo *FECHA* de la base de datos en el atributo *fecha* de la clase de entidad. Admite uno de los siguientes argumentos:

@Temporal(TemporalType.DATE)

Ignora la hora, quedando el campo acotado sólo a la fecha.

@Temporal(TemporalType.TIME)

Ignora la fecha, quedando el campo acotado sólo a la hora.

@Temporal(TemporalType.TIMESTAMP)

Tiene en cuenta la fecha y la hora.

@ManyToOne

Esta anotación establece una asociación de *muchos a uno*. En nuestro programa está registrando una asociación de muchas facturas a un solo cliente. Cada uno de los objetos *Factura* se relaciona con un solo objeto *Cliente* y lo hace a través del campo *NIF*.

```
//bi-directional many-to-one association to Cliente
```

```
@ManyToOne
```

```
@JoinColumn(name = "NIF")
```

```
private Cliente cliente;
```

A partir de un objeto *Factura* tenemos disponible toda la información del cliente al que pertenece, a partir del atributo *cliente*.

@JoinColumn(name = "NIF")

Acompaña una anotación de tipo *@ManyToOne* para indicar cuál es la columna que establece la asociación.

Las clases de entidad Detalle y DetallePK

La clave principal de la tabla *DETALLE* está compuesta por la unión de los campos *NUMERO* (número de factura) y *CODIGO* (código de producto).

Esta situación de clave primaria compuesta es un tanto especial dentro de las clases de entidad.

1. Comenzaremos analizando la clase *DETALLE*. Recordemos cómo está montado el código de la clase *Detalle*:

```
package model;

import java.io.Serializable;
import javax.persistence.*;

@Entity
@NamedQuery(name="Detalle.findAll", query="SELECT d FROM Detalle d")
public class Detalle implements Serializable {
    private static final long serialVersionUID = 1L;

    @EmbeddedId
    private DetallePK id;

    private float precio;

    private int unidades;

    //bi-directional many-to-one association to Factura
    @ManyToOne
    @JoinColumn(name="NUMERO")
    private Factura factura;

    //bi-directional many-to-one association to Producto
    @ManyToOne
    @JoinColumn(name="CODIGO")
    private Producto producto;

    public Detalle() {
    }

    public DetallePK getId() {
        return this.id;
    }

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

    public float getPrecio() {
        return this.precio;
    }

    public void setPrecio(float precio) {
        this.precio = precio;
    }
}
```

```
public int getUnidades() {
    return this.unidades;
}

public void setUnidades(int unidades) {
    this.unidades = unidades;
}

public Factura getFactura() {
    return this.factura;
}

public void setFactura(Factura factura) {
    this.factura = factura;
}

public Producto getProducto() {
    return this.producto;
}

public void setProducto(Producto producto) {
    this.producto = producto;
}

}
```

Ahora, vamos a analizar las partes más relevantes:

@EmbeddedId

Esta anotación, al igual que la anotación `@Id`, está indicando que el atributo que sigue es una clave primaria, pero además indica que no se trata de un dato elemental, sino de un objeto definido en otra clase, en este caso la clase `DetallePK`.

`@EmbeddedId`

```
private DetallePK id;
```

El atributo o propiedad `id` es un objeto de la clase `DetallePK`.

@ManyToOne

```
//bi-directional many-to-one association to Factura
```

```
@ManyToOne
```

```
@JoinColumn(name="NUMERO")
```

```
private Factura factura;
```

```
//bi-directional many-to-one association to Producto
```

```
@ManyToOne
```

```
@JoinColumn(name="CODIGO")
```

```
private Producto producto;
```

Cada objeto de la clase *Detalle* es un compuesto formado por la factura que incluye dicha línea de detalle, y el producto que se vende, además del resto de los atributos.

2. Vamos a analizar ahora el código de la clase *DetallePK*. Observa el código de la clase *DetallePK*:

```
package model;

import java.io.Serializable;
import javax.persistence.*;

@Embeddable
public class DetallePK implements Serializable {
    //default serial version id, required for serializable classes.
    private static final long serialVersionUID = 1L;

    @Column(insertable=false, updatable=false)
    private int numero;

    @Column(insertable=false, updatable=false)
    private String codigo;

    public DetallePK() {
    }
    public int getNumero() {
        return this.numero;
    }
    public void setNumero(int numero) {
        this.numero = numero;
    }
    public String getCodigo() {
        return this.codigo;
    }
    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }

    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (!(other instanceof DetallePK)) {
            return false;
        }
        DetallePK castOther = (DetallePK)other;
        return
            (this.numero == castOther.numero)
            && this.codigo.equals(castOther.codigo);
    }

    public int hashCode() {
        final int prime = 31;
        int hash = 17;
        hash = hash * prime + this.numero;
        hash = hash * prime + this.codigo.hashCode();
    }
}
```

```
        return hash;
    }
}
```

Vamos a comentar las partes más importantes:

@Embeddable

Esta anotación va delante de una clase que tiene como función embeber una clave primaria compuesta.

@Embeddable

```
public class DetallePK implements Serializable { ... }
```

Cada objeto de la clase *DetallePK* es una clave primaria embebida, que servirá de componente a la clase *Detalle*.

@Column(insertable=false, updatable=false)

La anotación *@Column* con los atributos *insertable* y *updatable* y con los valores *false*, provoca que el atributo que acompaña no sea tenido en cuenta en el momento en que el framework ORM autogenera las sentencias SQL *INSERT* y *UPDATE*, cuando haya que persistir el objeto.

```
@Column(insertable=false, updatable=false)
private int numero;
```

```
@Column(insertable=false, updatable=false)
private String codigo;
```

Los objetos de la clase *DetallePK* no están pensados para persistirse, sino para servir de componente dentro de un objeto *Detalle* que sí podría ser persistido (guardado físicamente en la base de datos). Por este motivo, las dos propiedades o atributos de la clase *DetallePK* llevan esta anotación.

Los métodos equals() y hashCode()

Observa que también se han sobrescrito los métodos *equals()* y *hashCode()*, métodos que vienen heredados de la superclase *Object*.

Recuerda que el método *hashCode()* devuelve el código *hash* asociado al objeto. El código *hashes* un identificador de 32 bits que identifica al objeto. Es posible sobrescribir el método *hashCode()* para personalizar la generación del identificador.

El método `equals()` compara el objeto actual con otro objeto y devuelve `true` si son iguales; `equals()` considera que dos métodos son iguales si devuelven el mismo código `hash`.

En la clase `DetallePK` se sobrescriben estos métodos para controlar la no existencia de clave primaria duplicada a la hora de persistir un objeto de la clase `Detalle`.

Si lo necesitas, puedes repasar los conceptos sobre este tema volviendo a la lección 6.2. *Generalización / especialización: herencia de la asignatura de programación; en concreto, en el apartado Los métodos `hashCode` y `equals`.*

Buscar información de un cliente

A continuación, veremos cómo obtener un objeto `Cliente` a través del método `find()` del `EntityManager`.

Recuerda que para trabajar con JPA necesitas un objeto de la clase `EntityManager`, que será el administrador de esa base de datos virtual orientada a objetos que se crea a partir de la base de datos real.

En nuestro caso, esa base de datos virtual orientada a objetos está formada por objetos de las clases `Cliente`, `Factura`, `Detalle` y `Producto`.

Nuestro `EntityManager` cuenta con un método `find()` que nos permite obtener un objeto `Cliente`, `Factura`, `Detalle` o `Producto` a partir de la clave primaria.

```
Cliente cli = em.find(Cliente.class, "43434343A");
```

Esta simple línea de código te permitirá obtener el objeto `Cliente` con `NIF = "43434343A"`. Si no existe ningún cliente con el NIF especificado, el método `find()` devuelve `null`.

Nuestro programa va creciendo. Es hora de mejorar la interfaz de usuario añadiendo un menú de *opciones*, para que se pueda indicar la opción a realizar en tiempo de ejecución.

```
GESTIÓN FERRETERÍA
-----
1. Inventario
2. Consultar todas las facturas
3. Información de un cliente
4. Añadir nuevo cliente
5. Añadir nueva factura
6. Listado de productos para reponer
7. Terminar programa
¿Qué opción eliges?
```

Este menú permitirá al usuario seleccionar la opción deseada o terminar escribiendo la opción 7.

Otro cambio que vamos a introducir es que sólo obtendremos el *EntityManager* una vez que estemos en el método *main* y no lo volveremos a obtener cada vez que invoquemos a cada uno de los métodos. Para ello, hemos tenido que declarar las variables *factoria* y *em* como propiedades privadas de clase para que estén disponibles en todos los métodos.

Por ahora, nuestro programa queda así:

```
import java.util.List;
import java.util.Scanner;

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

import model.Cliente;
import model.Detalle;
import model.Factura;
import model.Producto;

public class Principal {
    private static Scanner lector;
    private static EntityManagerFactory factoria;
    private static EntityManager em;

    public static void main(String[] args) {

        factoria =
Persistence.createEntityManagerFactory("PrimerProyectoJPA");
        em = factoria.createEntityManager();

        lector = new Scanner(System.in);
        String opcion;

        do {
            System.out.println();
            System.out.println();
            System.out.println("GESTIÓN FERRETERÍA");
            System.out.println("-----");
            System.out.println("1. Inventario");
            System.out.println("2. Consultar todas las facturas");
            System.out.println("3. Información de un cliente");
            System.out.println("4. Añadir nuevo cliente");
            System.out.println("5. Añadir nueva factura");
            System.out.println("6. Listado de productos para
reponer");
            System.out.println("7. Terminar programa");
            System.out.println("¿Qué opción eliges?");
            opcion = lector.nextLine();

            switch (opcion) {
                case "1": inventario(); break;
                case "2": facturasDetalles(); break;
                case "3": consultaCliente(); break;
                case "4": aniadirCliente(); break;
                case "5": aniadirFactura(); break;
                case "6": listadoReposición(); break;
            }
        } while (opcion != "7");
    }
}
```

```

        case "7": System.out.println("Hasta pronto");
    }

} while (!opcion.equals("7"));

lector.close();
}

public static void inventario() {
    TypedQuery<Producto> query =
em.createNamedQuery("Producto.findAll", Producto.class);
    List<Producto> productos = query.getResultList();

    for (Producto p : productos) {
        System.out.println(p.getCodigo() + " - " +
p.getDescripcion() + " - " + p.getPrecio() + " € - " + p.getStock() + " unidades.");
    }
}

public static void facturasDetalles() {
    TypedQuery<Factura> query =
em.createNamedQuery("Factura.findAll", Factura.class);
    List<Factura> facturas = query.getResultList();

    for (Factura f : facturas) {
        System.out.println(f.getNumero() + " - " + f.getFecha() +
" - " + f.getCliente().getNombre());
        for (Detalle d : f.getDetalles()) {
            Producto p = d.getProducto();
            System.out.println("    " + p.getCodigo() + " - " +
+ p.getDescripcion() + " - " + d.getPrecio() + "€ " + d.getUnidades());
        }
    }
}

public static void consultaCliente() {
    System.out.println("Escribe NIF del cliente buscado: ");
    String nif = lector.nextLine();
    Cliente cli = em.find(Cliente.class, nif);
    if (cli == null) {
        System.out.println("No se ha encontrado el cliente con
NIF = " + nif);
    }
    else {
        System.out.println("Nombre: " + cli.getNombre());
        System.out.println("Domicilio: " + cli.getDomicilio());
        System.out.println("Teléfono: " + cli.getTlf());
        System.out.println("Ciudad: " + cli.getCiudad());
        System.out.println("FACTURAS: ");
        for (Factura f : cli.getFacturas()) {
            System.out.println("    " + f.getNumero() + " " +
f.getFecha());
        }
    }
}

public static void aniadirCliente() {
}

```

```
public static void anadirFactura() {  
}  
  
public static void listadoReposición() {  
}  
}
```

Tenemos desarrolladas las tres primeras opciones y, para las demás, el método está preparado en vacío para ser implementado más adelante.

Añadir un cliente

Hemos montado la estructura de la aplicación, pero faltan varios métodos por implementar. Veamos ahora la implementación del método *aniadirCliente()*, que añadirá una nueva fila en la tabla *CLIENTE*.

Añadir un nuevo cliente es tan sencillo como construir un nuevo objeto de tipo *Cliente* y luego invocar al método *persist()* del *EntityManager*. Eso sí, lo haremos dentro de una transacción. En primer lugar, se solicitará al usuario que introduzca el NIF del nuevo cliente y se efectuará una búsqueda por medio del método *find()*. Si el cliente existe, informaremos al usuario, y si no existe, le daremos de alta. En este caso, se solicitará al cliente que introduzca por teclado el resto de los datos.

```
public static void anadirCliente() {  
  
    System.out.println("NIF del nuevo cliente: ");  
    String nif = lector.nextLine();  
    Cliente cli = em.find(Cliente.class, nif);  
    if (cli == null) {  
        cli = new Cliente();  
        cli.setNif(nif);  
        System.out.println("Nombre: ");  
        cli.setNombre(lector.nextLine());  
        System.out.println("Dirección: ");  
        cli.setDomicilio(lector.nextLine());  
        System.out.println("Teléfono: ");  
        cli.setTlf(lector.nextLine());  
        System.out.println("Ciudad: ");  
        cli.setCiudad(lector.nextLine());  
  
        EntityTransaction et = em.getTransaction();  
        et.begin();  
        em.persist(cli);  
        et.commit();  
    }  
    else {  
        System.out.println("Ya existe un cliente con NIF: " + nif);  
    }  
}
```

Hemos creado un nuevo objeto *Cliente* cuya referencia está en la variable *cli*. Luego, **para persistir el nuevo cliente, hemos utilizado las siguientes líneas de código:**

EntityTransaction et = em.getTransaction();

A través del objeto *EntityManager* (*em*) obtenemos un objeto de tipo *EntityTransaction*, que nos servirá para gestionar las transacciones en las operaciones de actualización de la base de datos.

et.begin();

Inicia la transacción.

em.persist(cli);

Persiste el nuevo cliente en la base de datos.

et.commit();

Finaliza la transacción, realizando los cambios.

También se podría invocar al método *rollback()* para revertir estos cambios.

Las operaciones CRUD con JPA

CRUD (Create, Read, Update y Delete) hace referencia a las operaciones básicas que pueden realizarse sobre las filas de una tabla dentro de una base de datos relacional: crear, leer, actualizar y borrar.

Nuestro objeto *EntityManager* (*em* para nuestro programa) te permite realizar las cuatro operaciones CRUD a través de las clases de entidad, utilizando los siguientes métodos:

- ***persist(obj)***: persiste el objeto indicado en el argumento.
- ***find(entityClass, primaryKey)***: busca el valor especificado como *primaryKey* y devuelve el objeto encontrado, cuyo tipo viene especificado en el primer argumento.
- ***merge(obj)***: actualiza la fila de la base de datos, con los valores del objeto especificado como argumento.
- ***delete(obj)***: elimina de la base de datos la fila que corresponde, con el objeto pasado como argumento.

1. Crear nueva fila

```
cli = new Cliente();
cli.setNif("12345678A");
cli.setNombre("PERICO DE LOS PALOTES");
cli.setDomicilio("C/ PERICO, 45");
```

```
cli.setTlf("612345678");
cli.setCiudad("MADRID");

EntityTransaction et = em.getTransaction();
et.begin();
em.persist(cli);
et.commit();
```

2. Leer una fila

```
Cliente cli = em.find(Cliente.class, "43434343A");
System.out.println(cli.getNombre());
```

3. Actualizar una fila

```
Cliente cli = em.find(Cliente.class, "43434343A");
EntityTransaction et = em.getTransaction();
et.begin();
cli.setDomicilio("C/ Nueva, 25");
em.merge(cli);
et.commit();
```

4. Eliminar una fila

```
Cliente cli = em.find(Cliente.class, "43434343A");
EntityTransaction et = em.getTransaction();
et.begin();
em.remove(cli);
et.commit();
```

Java Persistence Query Language (JPQL)

JPQL es un lenguaje con una estructura muy similar a SQL, pero adaptado a la manipulación de objetos de entidad.

La gestión de las consultas JPQL será responsabilidad de un objeto *Query* o de un objeto *TypedQuery*.

Veamos el formato de la sentencia *SELECT* en JPQL:

```
SELECT atributos FROM ClaseEntidad Alias
WHERE criterio
GROUP BY atributos
```

```
HAVING criterio  
ORDER BY atributos
```

En el *criterio* se puede utilizar cualquiera de los operadores de SQL.

Hay que tener en cuenta que la consulta no se realiza sobre la base de datos real, sino sobre la base de datos orientada a objetos virtual gestionada por el *EntityManager*.

Esto significa que todos los atributos hacen referencia a las propiedades de las clases de entidad.

Veamos algunos ejemplos:

Ejemplo 1:

```
SELECT cli FROM Cliente cli
```

Selecciona todos los clientes. El identificador *cli* es el alias que hace referencia a cada objeto *Cliente* obtenido.

Ejemplo 2:

```
SELECT cli FROM Cliente cli WHERE cli.ciudad = 'MADRID'
```

Selecciona todos los clientes de *Madrid* y devuelve una colección de objetos *Cliente*.

Ejemplo 3:

```
SELECT cli FROM Cliente cli WHERE cli.nombre LIKE '%PEREZ%'
```

Selecciona todos los clientes que contengan el apellido *PEREZ* en cualquier posición y devuelve una colección de objetos *Cliente*.

Ejemplo 4:

```
SELECT cli.nombre FROM Cliente cli WHERE cli.nombre LIKE '%PEREZ%'
```

Selecciona el nombre de todos los clientes que contengan el apellido *PEREZ* en cualquier posición. Devuelve el resultado en una sola columna y puede recogerse como una colección de objetos *String*.

Ejecutar una consulta JPQL de selección desde Java

Para ejecutar una consulta JPQL de selección desde Java necesitamos obtener un objeto *TypedQuery* a través del método *createQuery()* del *EntityManager*. Luego, debemos invocar al método ***getResultSet()*** del objeto *TypedQuery*.

```
TypedQuery<Cliente> query = em.createQuery("SELECT cli FROM Cliente cli WHERE cli.nombre LIKE '%PEREZ%'", Cliente.class);
List<Cliente> clientes = query.getResultList();
for (Cliente c : clientes) {
    System.out.println(c.getNombre() + " - " + c.getTlf());
}
```

La consulta anterior obtiene una colección de objetos *Cliente*, pero si sólo necesitamos una columna, podríamos obtener una colección de objetos *String*. Observa el siguiente ejemplo:

```
TypedQuery<String> query = em.createQuery("SELECT DISTINCT cli.ciudad FROM Cliente cli", String.class);
List<String> ciudades = query.getResultList();
for (String c : ciudades) {
    System.out.println(c);
}
```

La consulta devuelve una sola columna, así que puede ser recogida en una colección de tipo *String*.

1. Consulta que devuelve un solo valor

Para ejecutar una consulta que estamos seguros de que sólo devuelve un objeto y no una colección de ellos, utilizamos el método ***getSingleResult()*** en lugar del método *getResultList()*. A continuación vamos a probar con una consulta que devuelve el último número de factura:

```
TypedQuery<Integer> query = em.createQuery("SELECT MAX(f.numero) FROM Factura f", Integer.class);
Integer maxNumFactura = query.getSingleResult();
System.out.println("Último número de factura: " + maxNumFactura);
```

La consulta devuelve un único valor de tipo *Integer*.

2. Consulta que devuelve una colección de arrays

La mayor parte de las consultas JPQL que ejecutamos devolverán una colección de objetos de un tipo específico: *Cliente*, *Factura*, *Detalle* o *Producto*. Pero cuando especificamos un número determinado de columnas que ya no se corresponde con la estructura de ninguna de las clases de entidad, podemos recoger el resultado como una colección de arrays. Veamos un ejemplo:

```
TypedQuery<Object[]> query = em.createQuery("SELECT pro.codigo, pro.descripcion, pro.minimo-pro.stock FROM Producto pro WHERE pro.stock < pro.minimo", Object[].class);
List<Object[]> resultados = query.getResultList();
for (Object[] p : resultados) {
    System.out.println(p[0] + " - " + p[1] + " - " + p[2]);
}
```

Esta consulta selecciona los productos que hay que reponer y devuelve una colección de arrays de tres elementos: *codigo*, *descripcion* y una columna calculada: *minimo-stock*.

3. Consulta de acción

También es posible ejecutar consultas de acción con JPQL (*UPDATE*, *INSERT* o *DELETE*). Para ello, necesitamos obtener un objeto de tipo *Query* y ejecutar la consulta con el método *executeUpdate()*, que devolverá un número entero con la cantidad de objetos o filas afectadas.

```
EntityTransaction et = em.getTransaction();
et.begin();
Query query = em.createQuery("UPDATE Cliente c SET c.domicilio = 'C/ LUNA, 25' WHERE c.nif='43434343A'");
int afectados = query.executeUpdate();
et.commit();
System.out.println("Objetos afectados: " + afectados);
```

Cambia la dirección del cliente con NIF = "43434343A".

4. Consulta con parámetros

JPQL permite declarar sentencias con parámetros que pueden variar en tiempo de ejecución. Un parámetro se define con una interrogación seguida de un número que identifica la posición del parámetro. El parámetro se establece con el método *setParameter(posición, valor)*. Veamos un ejemplo:

```
String sql = "SELECT pro FROM Producto pro WHERE pro.codigo = ?1";
TypedQuery<Producto> query = em.createQuery(sql, Producto.class);
query.setParameter(1, "TOR7");
Producto p = query.getSingleResult();
System.out.println(p.getDescripcion() + " - " + p.getPrecio()+"€");
```

Obtiene el objeto *Producto* que corresponde con el producto con código *T0R7*.

Listado de productos para reponer

A continuación vamos a implementar el método *listadoReposición()*, que mostrará un listado con los productos pendientes de reposición y las unidades a reponer.

Recuerda que un producto necesita reposición si tiene un valor mayor en el campo *minimo* que en el campo *stock*.

```
public static void listadoReposición() {  
    TypedQuery<Object[]> query = em.createQuery("SELECT pro.codigo,  
pro.descripcion, pro.minimo-pro.stock FROM Producto pro WHERE pro.stock <  
pro.minimo", Object[].class);  
    List<Object[]> resultados = query.getResultList();  
    for (Object[] p : resultados) {  
        System.out.println(p[0] + " - " + p[1] + " - " + p[2]);  
    }  
}
```

Este tipo de consulta la estudiamos en el apartado anterior y ahora la hemos integrado en la aplicación final.

Añadir una factura

Vamos a abordar ahora la tarea más laboriosa de nuestro programa: dar de alta una nueva factura implementando el método *añadirFactura()*.

No es que añadir una nueva factura en la base de datos sea complicado. En realidad, no haremos nada que no hayamos estudiado ya en los apartados anteriores. Resulta laborioso porque implica varias tareas:

- Buscar al cliente al que pertenece la factura.
- Añadir la factura.
- Añadir las líneas de detalle de la factura, lo que implica también localizar los productos que van a venderse.

Veamos el código para el método *añadirFactura()* y después lo analizaremos paso por paso.

```
public static void añadirFactura() {  
    System.out.println("NIF del cliente: ");
```

```

String nif = lector.nextLine();
Cliente cli = em.find(Cliente.class, nif);
if (cli==null) {
    System.out.println("No existe el cliente con NIF = " + nif);
    System.out.println("Seleccione primero la opción 4");
}
else {
    System.out.println("Cliente: " + cli.getNombre());
    TypedQuery<Integer> query = em.createQuery("SELECT MAX(f.numero)
FROM Factura f", Integer.class);
    Integer numFactura = query.getSingleResult() + 1;
    System.out.println("Número de factura: " + numFactura);

    // Construimos objeto factura.
    Factura f = new Factura();
    f.setNumero(numFactura);
    f.setCliente(cli);
    f.setPagado(false);
    f.setFecha(new Date());
    // Añadimos la factura al cliente.
    cli.addFactura(f);

    // Iniciamos la transacción y persistimos la factura.
    EntityTransaction et = em.getTransaction();
    et.begin();
    em.persist(f);

    // Construimos los detalles de la factura.
    String continuar;
    do {
        System.out.println("Código de artículo: ");
        String codigo = lector.nextLine();
        Producto pro = em.find(Producto.class, codigo);
        if (pro==null) {
            System.out.println("No se encuentra el producto
con código " + codigo);
        }
        else {
            System.out.println("Descripción: " +
pro.getDescripcion());
            System.out.println("Precio: " + pro.getPrecio());
            System.out.println("¿Cuantas unidades desea? ");
            int unidades = lector.nextInt();
            lector.nextLine(); // Recoge el retorno de carro.

            // Creamos el objeto Detalle con su primary key
            compuesta.
            DetallePK pk = new DetallePK();
            pk.setCodigo(codigo);
            pk.setNumero(numFactura);
            Detalle d = new Detalle();
            d.setId(pk);
            d.setPrecio(pro.getPrecio());
            d.setUnidades(unidades);
            d.setProducto(pro);
            d.setFactura(f);

            // Persistimos el detalle.
            em.persist(d);

            // Añadimos el detalle a la factura.
    }
}

```

```

        f.addDetalle(d);
    }
    System.out.println("¿Deseas seguir comprando (S/N)? ");
    continuar = lector.nextLine();
} while (continuar.toUpperCase().equals("S"));

et.commit();
}
}

```

Vamos a dividir el código en partes para analizarlo poco a poco:

```

public static void anadirFactura() {
    System.out.println("NIF del cliente: ");
    String nif = lector.nextLine();
    Cliente cli = em.find(Cliente.class, nif);
    if (cli==null) {
        System.out.println("No existe el cliente con NIF = " + nif);
        System.out.println("Seleccione primero la opción 4");
    }
    else {
        .....
    }
}

```

En primer lugar, solicitamos al usuario que teclee el NIF del cliente al que se le emitirá la factura y realizaremos una búsqueda para obtener el objeto *Cliente*.

- **Si no existe el cliente**, se informará al usuario, sugiriéndole que primero elija la opción 4 del menú (*Añadir nuevo cliente*).
- **Si el cliente existe**, continuaremos con el resto de los pasos hasta dar de alta la nueva factura.

```

System.out.println("Cliente: " + cli.getNombre());
TypedQuery<Integer> query = em.createQuery("SELECT MAX(f.numero) FROM Factura f", Integer.class);
Integer numFactura = query.getSingleResult() + 1;
System.out.println("Número de factura: " + numFactura);

```

Una vez que hemos obtenido el objeto *Cliente*, obtenemos el número de la nueva factura a través de una consulta.

Los números de factura van consecutivos, así que **la nueva factura tendrá como número el mayor de los números de factura más una unidad**.

```
// Construimos objeto factura.
Factura f = new Factura();
```

```
f.setNumero(numFactura);
f.setCliente(cli);
f.setPagado(false);
f.setFecha(new Date());
// Añadimos la factura al cliente.
cli.addFactura(f);
```

Construimos un nuevo objeto *Factura* y se lo añadimos al cliente por medio del método *addFactura()*.

```
// Iniciamos la transacción y persistimos la factura.
EntityTransaction et = em.getTransaction();
et.begin();
em.persist(f);
```

Iniciamos la transacción y persistimos la nueva factura. Pero no cerramos después la transacción porque todavía falta construir y persistir las líneas de detalle (objetos *Detalle*).

```
// Construimos los detalles de la factura.
String continuar;
do {
    System.out.println("Código de artículo: ");
    String codigo = lector.nextLine();
    Producto pro = em.find(Producto.class, codigo);
    if (pro==null) {
        System.out.println("No se encuentra el producto con código " +
codigo);
    }
    else {
        System.out.println("Descripción: " + pro.getDescripcion());
        System.out.println("Precio: " + pro.getPrecio());
        System.out.println("¿Cuantas unidades desea? ");
        int unidades = lector.nextInt();
        lector.nextLine(); // Recoge el retorno de carro.

        // Creamos el objeto Detalle con su primary key compuesta.
        DetallePK pk = new DetallePK();
        pk.setCodigo(codigo);
        pk.setNumero(numFactura);
        Detalle d = new Detalle();
        d.setId(pk);
        d.setPrecio(pro.getPrecio());
        d.setUnidades(unidades);
        d.setProducto(pro);
        d.setFactura(f);

        // Persistimos el detalle.
        em.persist(d);

        // Añadimos el detalle a la factura.
        f.addDetalle(d);
    }
    System.out.println("¿Deseas seguir comprando (S/N) ? ");
    continuar = lector.nextLine();
```

```
} while (continuar.toUpperCase().equals("S"));
```

De manera repetitiva, mientras el usuario quiera, permitiremos añadir nuevas líneas de detalle para la factura. Por cada detalle se solicita al usuario el código de producto y se busca. Una vez localizado el producto, se informa de la descripción y el precio, se solicitan las unidades que se van a vender y, a continuación, se crea la nueva línea de detalle y se persiste.

Cada línea de detalle se añade al objeto *Factura* mediante el método *addDetalle()*.

Si has ejecutado el programa, puede que no haya funcionado y que hayas terminado con una excepción de tipo *NullPointerException*. No te preocupes, porque lo vamos a solucionar.

De nuevo es un problema que viene originado por cómo están implementadas las clases de entidad, y tendremos que hacer un pequeño retoque. Recuerda que un objeto *Factura* incluye una colección de tipo *List* de objetos *Detalle*.

```
@OneToOne (mappedBy="factura")
    private List<Detalle> detalles;
```

Cuando se construye un objeto *Factura* a partir de la consulta de una factura existente, la propiedad *detalles* es inyectada con la colección de detalles de dicha factura, y se podrían añadir más *detalle* mediante la llamada al método *addDetalle()*.

Pero cuando se trata de una factura añadida recientemente que todavía no tiene líneas de detalle, la propiedad *detalles* tiene el valor *null* y el método *addDetalle()* arroja una excepción, porque intenta añadir un elemento a una colección que no está construida.

La solución está en **añadir una línea al constructor de *Factura* para crear la colección**.

¡Ojo! Recuerda que *List* no es una clase sino una interfaz, y es necesario construir un objeto de una clase que implemente la interfaz *List*; por lo tanto, construiremos un objeto *Vector*.

```
public Factura() {
    detalles = new Vector<Detalle>();
}
```

Constructor de *Factura* añade un nuevo objeto *Vector* vacío, listo para guardar nuevos objetos *Detalle*.

Podríamos tener el mismo problema para añadir nuevas facturas a un objeto *Cliente* si el cliente no tiene aún una factura. También deberíamos modificar el construir de *Cliente*, dejándolo así:

```
public Cliente() {  
    facturas = new Vector<Factura>();  
}
```

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- **JPA (Java Persistence API)** es el *framework* estándar proporcionado por *Java Enterprise Edition* (Java EE) para la capa de persistencia que implementa el concepto de ORM.
- Un **framework ORM** implementa la técnica de **Mapeo objeto-relacional**, proporcionando una estructura de objetos que representa la base de datos física.
- El Mapeo objeto-relacional se efectúa por medio de las **clases de entidad**, que representan la estructura de tablas de la base de datos real. Dentro de nuestra aplicación, trabajamos con una base de datos orientada a objetos que es una representación de la base de datos real.
- La clase principal con la que trabajamos en una aplicación JPA es el **EntityManager**, administrador de los objetos de entidad (base de datos orientada a objetos virtual) que nos permite realizar las operaciones CRUD con la base de datos, a partir de los objetos de entidad.
- En un proyecto JPA, la configuración de conexión a una base de datos está dentro del archivo **persistence.xml**.

4.1. Representación de datos XML (CSS, XSL)



Índice

Objetivos	3
Documentos XML	4
Formato de los documentos XML	4
¿Cómo se ven en el navegador?.....	6
Aplicar estilo al documento XML	11
Aplicar hojas de estilo CSS	11
Trasformaciones XSL	13
Usar XML, XSL y CSS.....	21
Filtrar los resultados con XPath.....	22
Despedida	27
Resumen.....	27

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Conocer la estructura de un documento XML.
- Construir documentos XML.
- Añadir estilo estético a la salida de los documentos XML, utilizando hojas de estilo CSS.
- Añadir estilo estético a la salida de los documentos XML, utilizando trasformaciones XSL (*Extensible Stylesheet Language*).
- Realizar filtros utilizando XPath.

Documentos XML

Formato de los documentos XML

Los **ficheros XML** (*Extended Markup Language*) proveen un formato ligero para el intercambio de datos, lo que resulta especialmente interesante en aplicaciones web.

Los documentos SGML siguen un esquema jerárquico, compuesto por etiquetas con apertura y cierre que tienen la siguiente estructura: `<etiqueta> contenido </etiqueta>`. Cada etiqueta puede portar otras subetiquetas, formando así la jerarquía.

HTML y XML se basan en el estándar SGML. Tienen su origen en el lenguaje SGML (*Standard Generalised Mark-up Language*) definido por la norma ISO 8879 en 1986. Se trata de un estándar para lenguajes de marcado, con el objetivo de definir documentos independientes de las plataformas hardware y software.

Una etiqueta en un documento XML y representa un nodo o elemento que puede estar compuesto por otros nodos.

En el siguiente ejemplo tenemos un documento que representa los datos de varios cruceros.

```
<?xml version="1.0" encoding="UTF-8"?>
<cruceros>
    <crucero codigo="CRUMED21">
        <destino>Mediterraneo (Grecia, Italia)</destino>
        <detalles>
            <cia>Costa cruceros</cia>
            <dias>6 días</dias>
            <fechaSalida>2018-12-26</fechaSalida>
        </detalles>
        <escalas>
            <escala dia="1">
                <parada>Venezia</parada>
                <llegada></llegada>
                <salida>18:00</salida>
            </escala>
            <escala dia="2">
                <parada>Navegación</parada>
                <llegada></llegada>
                <salida></salida>
            </escala>
            <escala dia="3">
                <parada>Agostini</parada>
                <llegada>7:00</llegada>
                <salida>14:00</salida>
            </escala>
            <escala dia="4">
                <parada>Santorini</parada>
                <llegada>9:00</llegada>
                <salida>20:00</salida>
            </escala>
            <escala dia="5">
```

```
<parada>Bari</parada>
<llegada>8:00</llegada>
<salida>14:00</salida>
</escala>
<escala dia="6">
    <parada>Venecia</parada>
    <llegada>8:30</llegada>
    <salida></salida>
</escala>
</escalas>
</crucero>
<crucero codigo="CRUATL22">
    <destino>Atlántico (España, Marruecos, Portugal)</destino>
    <detalles>
        <cia>MSC Cruceros</cia>
        <dias>7 días</dias>
        <fechaSalida>2019-02-13</fechaSalida>
    </detalles>
    <escalas>
        <escala dia="1">
            <parada>Barcelona</parada>
            <llegada></llegada>
            <salida>18:00</salida>
        </escala>
        <escala dia="2">
            <parada>Navegación</parada>
            <llegada></llegada>
            <salida></salida>
        </escala>
        <escala dia="3">
            <parada>Casablanca</parada>
            <llegada>7:00</llegada>
            <salida>22:00</salida>
        </escala>
        <escala dia="4">
            <parada>Navegación</parada>
            <llegada></llegada>
            <salida></salida>
        </escala>
        <escala dia="5">
            <parada>Santa Cruz de Tenerife</parada>
            <llegada>9:00</llegada>
            <salida>16:00</salida>
        </escala>
        <escala dia="6">
            <parada>Funchal</parada>
            <llegada>8:00</llegada>
            <salida>19:00</salida>
        </escala>
        <escala dia="7">
            <parada>Barcelona</parada>
            <llegada>17:00</llegada>
            <salida></salida>
        </escala>
    </escalas>
</crucero>
</cruceros>
```

La primera línea del documento (cabecera), es la que identifica el documento como tipo XML y donde se especifica la versión, tipo de codificación, etc. Luego va la apertura del nodo raíz que encierra el resto de los nodos.

```
<?xml version="1.0" encoding="UTF-8"?>
<cruceros>
    .....
</cruceros>
```

El documento del ejemplo está concebido para portar la información de varios cruceros, por lo que el nodo raíz `<cruceros>` contendrá un conjunto de nodos `<cruero>`. Ésta podría ser la estructura para una base de datos documental, donde cada etiqueta `<cruero>` encierra la información necesaria para la redacción de un documento que describe un crucero.

Los documentos XML, además de utilizarse para portar datos, sirven para establecer valores de configuración en las aplicaciones web. Un ejemplo es el archivo `persistence.xml` cuya función vimos anteriormente.

¿Cómo se ven en el navegador?

Los documentos XML están concebidos para portar datos, pero no la presentación o el diseño de los mismos.

En este apartado vamos a ver cómo se muestran los documentos XML en los principales navegadores.

Te proponemos que crees el documento XML dentro de un proyecto Java. Aunque no vamos a programar en Java, sí lo haremos en la siguiente lección, de modo que ya tendremos el proyecto preparado.

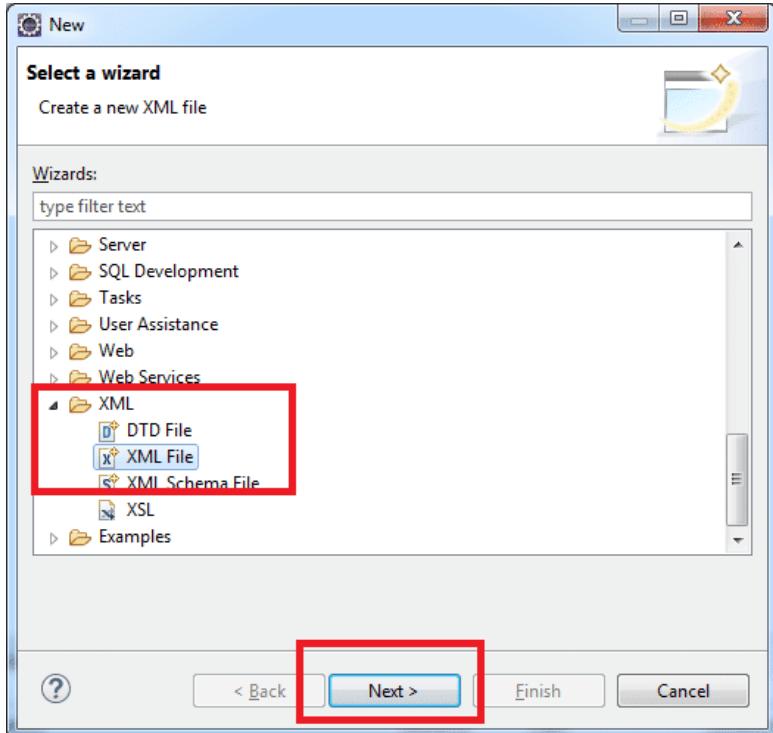
Sigue estos pasos:

1. Crea un proyecto Java estándar:

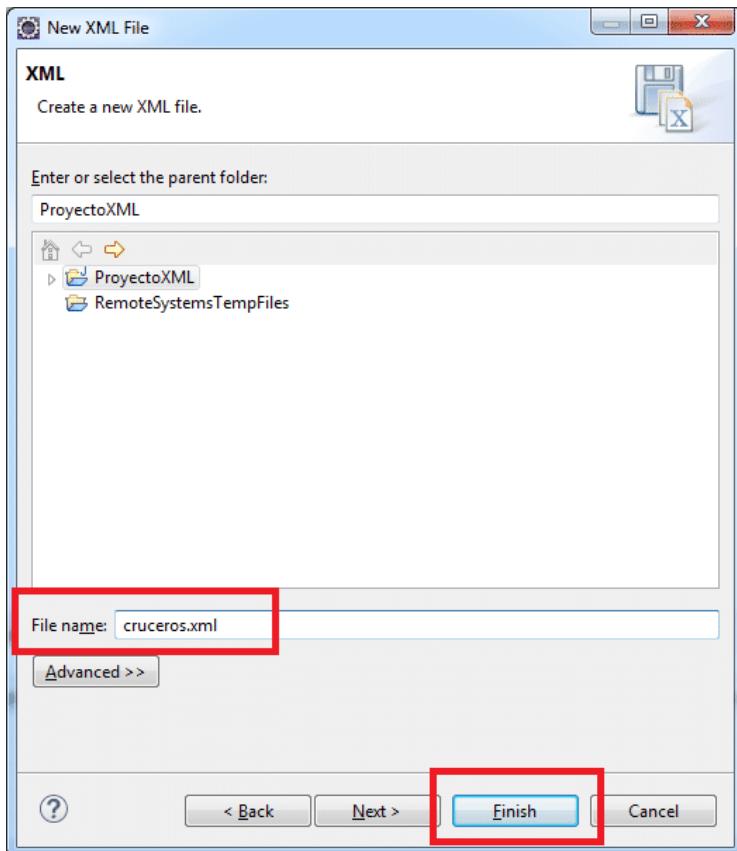
- Cambia a la perspectiva Java.
- Selecciona en el menú `File / New / Java project`.
- Escribe `ProyectoXML` como nombre del proyecto y pulsa `Finish`.

2. Crea un nuevo archivo XML dentro del proyecto:

- Haz clic derecho sobre el nombre del proyecto y selecciona `New / Other` en el menú contextual.
- Ahora te encuentras en el cuadro de diálogo `New`. Despliega la carpeta `XML`, escoge la opción `XML File` y pulsa `Next`.



- Escribe *cruceros.xml* como nombre del archivo y pulsa *Finish*.



3. Copia y pega el contenido del archivo XML desde aquí:

```
<?xml version="1.0" encoding="UTF-8"?>
<cruceros>
    <crucero codigo="CRUMED21">
        <destino>Mediterraneo (Grecia, Italia)</destino>
        <detalles>
            <cia>Costa cruceros</cia>
            <dias>6 días</dias>
            <fechaSalida>2018-12-26</fechaSalida>
        </detalles>
        <escalas>
            <escala dia="1">
                <parada>Venecia</parada>
                <llegada></llegada>
                <salida>18:00</salida>
            </escala>
            <escala dia="2">
                <parada>Navegación</parada>
                <llegada></llegada>
                <salida></salida>
            </escala>
            <escala dia="3">
                <parada>Agostini</parada>
                <llegada>7:00</llegada>
                <salida>14:00</salida>
            </escala>
            <escala dia="4">
                <parada>Santorini</parada>
                <llegada>9:00</llegada>
                <salida>20:00</salida>
            </escala>
            <escala dia="5">
                <parada>Bari</parada>
                <llegada>8:00</llegada>
                <salida>14:00</salida>
            </escala>
            <escala dia="6">
                <parada>Venecia</parada>
                <llegada>8:30</llegada>
                <salida></salida>
            </escala>
        </escalas>
    </crucero>
    <crucero codigo="CRUATL22">
        <destino>Atlántico (España, Marruecos, Portugal)</destino>
        <detalles>
            <cia>MSC Cruceros</cia>
            <dias>7 días</dias>
            <fechaSalida>2019-02-13</fechaSalida>
        </detalles>
        <escalas>
            <escala dia="1">
                <parada>Barcelona</parada>
                <llegada></llegada>
                <salida>18:00</salida>
            </escala>
            <escala dia="2">
                <parada>Navegación</parada>
```

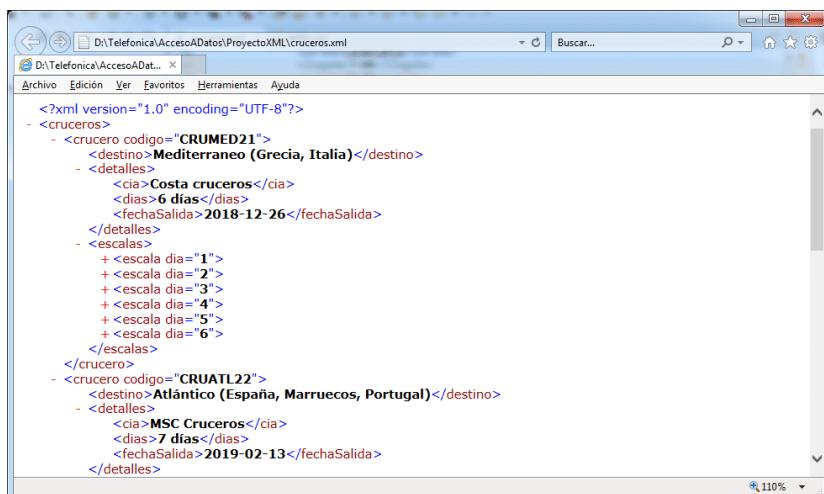
```

        <llegada></llegada>
        <salida></salida>
    </escala>
    <escala dia="3">
        <parada>Casablanca</parada>
        <llegada>7:00</llegada>
        <salida>22:00</salida>
    </escala>
    <escala dia="4">
        <parada>Navegación</parada>
        <llegada></llegada>
        <salida></salida>
    </escala>
    <escala dia="5">
        <parada>Santa Cruz de Tenerife</parada>
        <llegada>9:00</llegada>
        <salida>16:00</salida>
    </escala>
    <escala dia="6">
        <parada>Funchal</parada>
        <llegada>8:00</llegada>
        <salida>19:00</salida>
    </escala>
    <escala dia="7">
        <parada>Barcelona</parada>
        <llegada>17:00</llegada>
        <salida></salida>
    </escala>
</escalas>
</cruero>
</cruceros>

```

Con ayuda del explorador de archivos de Windows, accede a la carpeta del proyecto y abre el archivo *cruceros.xml* desde Internet Explorer.

Para lograrlo, haz clic derecho sobre el nombre del archivo en el explorador de Windows y luego selecciona "**Abrir con / Internet Explorer**".



Vista de un documento XML con Internet Explorer. Delante de los nodos compuestos puedes apreciar que aparecen los signos - (contraer) y + (expandir).

Répite la operación con los navegadores que tengas instalados en tu equipo:

```

<crueros>
  <cruero codigo="CRUMED21">
    <destino>Mediterraneo (Grecia, Italia)</destino>
    <detalles>
      <cia>Costa cruceros</cia>
      <dias>6 días</dias>
      <fechaSalida>2018-12-26</fechaSalida>
    </detalles>
    <escalas>
      <escala dia="1">...</escala>
      <escala dia="2">...</escala>
      <escala dia="3">
        <parada>Agostini</parada>
        <llegada>7:00</llegada>
        <salida>14:00</salida>
      </escala>
      <escala dia="4">...</escala>
      <escala dia="5">...</escala>
      <escala dia="6">...</escala>
    </escalas>
  </cruero>
  <cruero codigo="CRUATL22">
    <destino>Atlántico (España, Marruecos, Portugal)</destino>
    <detalles>
      <cia>MSC Cruceros</cia>
    </detalles>
  </cruero>

```

Vista de un documento XML con Google Chrome. Observa cómo delante de los nodos compuestos están los iconos de flecha hacia abajo (contraer) y flecha hacia la derecha (expandir).

```

<crueros>
  - <cruero codigo="CRUMED21">
    <destino>Mediterraneo (Grecia, Italia)</destino>
    - <detalles>
      <cia>Costa cruceros</cia>
      <dias>6 días</dias>
      <fechaSalida>2018-12-26</fechaSalida>
    </detalles>
    - <escalas>
      + <escala dia="1"></escala>
      - <escala dia="2">
        <parada>Navegación</parada>
        <llegada/>
        <salida/>
      </escala>
      + <escala dia="3"></escala>
      + <escala dia="4"></escala>
      + <escala dia="5"></escala>
      + <escala dia="6"></escala>
    </escalas>
  </cruero>
  - <cruero codigo="CRUATL22">
    <destino>Atlántico (España, Marruecos, Portugal)</destino>
    - <detalles>
      <cia>MSC Cruceros</cia>
      <dias>7 días</dias>
      <fechaSalida>2010-02-13</fechaSalida>
    </detalles>
  </cruero>

```

Vista de un documento XML con Firefox. Igual que en Internet Explorer, puedes apreciar que aparecen los signos - (contraer) y + (expandir).

Aplicar estilo al documento XML

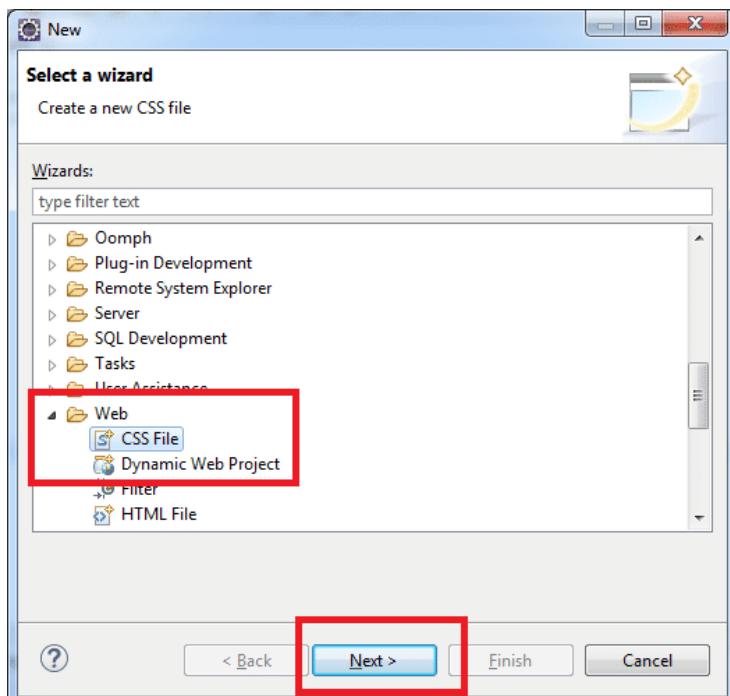
Aplicar hojas de estilo CSS

En este apartado aprenderás a **aplicar diseño estético al contenido de tu documento XML** por medio de una hoja de estilo CSS.

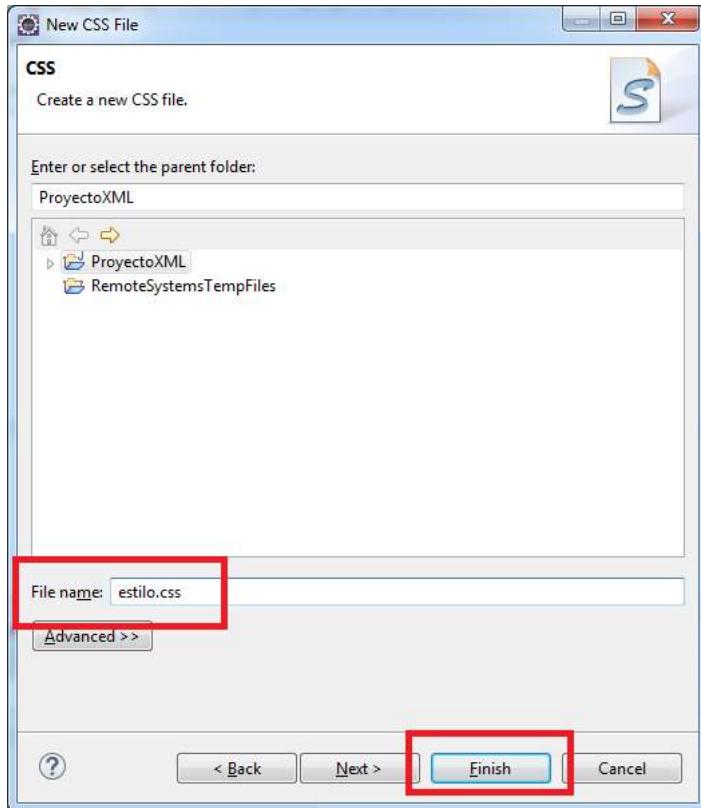
Sigue estos pasos:

1. Crea el archivo de hoja de estilo en el mismo proyecto donde tienes el documento XML

- Abre el proyecto Eclipse que contiene el archivo *cruceros.xml* si no lo tienes ya abierto.
- Haz clic derecho sobre el nombre del proyecto y selecciona en el menú contextual New / Other.
- Ahora te encuentras en el cuadro de diálogo New. Abre la carpeta Web, selecciona CSS File y luego pulsa el botón Next.



- Escribe *estilo.css* como nombre de archivo y pulsa *Finish*.



- Ahora, asigna un estilo a cada una de las etiquetas que forman parte del documento *cruceros.xml*. Copia y pega el código CSS que encontrarás a continuación.

```
@charset "ISO-8859-1";

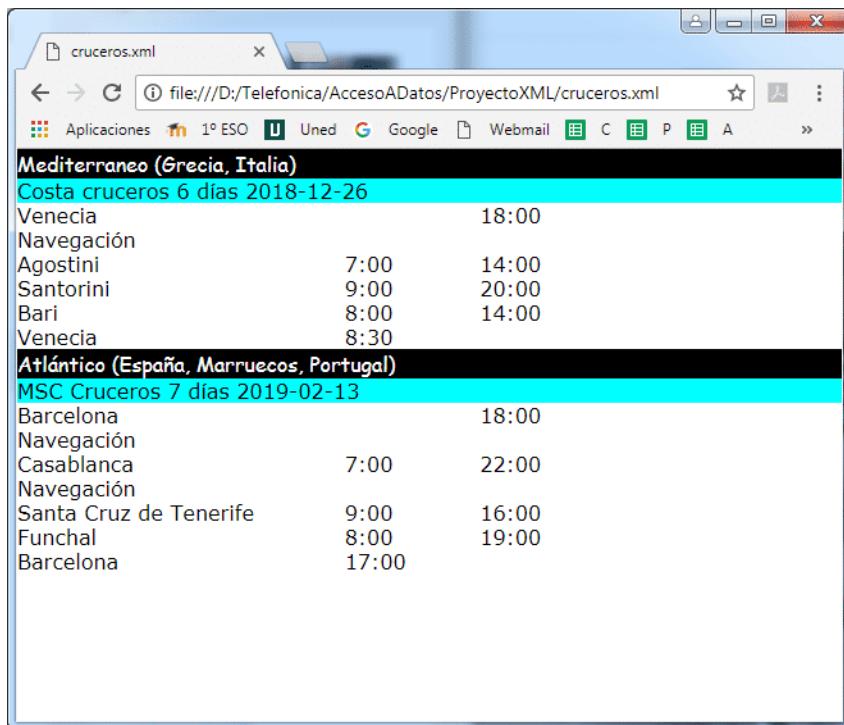
destino {
    background-color: black;
    color: white;
    font-family: "Comic sans ms";
    font-size: 30;
    display: block;
}
detalles {
    background-color: cyan;
    font-family: "Verdana";
    font-size: 20;
    display: block;
}
escala {
    font-family: "Verdana";
    font-size: 15;
    display: block;
}
parada {
    width: 250px;
    display: inline-block;
}
llegada, salida {
    width: 100px;
    display: inline-block;
}
```

2. Enlaza el archivo *cruceros.xml* con la hoja de estilos

- Para enlazar un documento XML con un archivo de hoja de estilos, se añade la etiqueta especial `<?xml-stylesheet type="text/css" href="estilo.css" ?>`, tal como verás a continuación.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="estilo.css" ?>
<cruceros>
.....
....
```

Ya puedes abrir tu documento *cruceros.xml* con cualquier navegador. El resultado será algo así:



Ya sabes cómo aplicar hojas de estilo CSS. ¿Continuamos con una nueva técnica?

Trasformaciones XSL

A continuación vamos a mostrarte la técnica denominada **Trasformaciones XSL (Extensible Stylesheet Language)**.

Esta técnica está formada por un conjunto de tres lenguajes. Juntos son capaces de trasformar la información contenida en los archivos XML para que pueda ser presentada al usuario en otros formatos, como, por ejemplo, HTML o PDF.

- XSLT (EXtensible Stylesheet Language Transformations)**: permite convertir documentos XML a otros formatos (por ejemplo, HTML).

- **XSL-FO (XSL Formatting Objects)**: permite especificar cómo se presentarán los datos, es decir, el diseño. Se utiliza principalmente para generar documentos PDF a partir de documentos XML.
- **XPath (XML Path Language)**: permite filtrar la información contenida en un documento XML.

Los documentos de transformación XSL se guardan en archivos con extensión .xsl y tienen la estructura de una página web HTML, pero encerrada dentro de una plantilla como la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <!-- AQUÍ VA LA ESTRUCTURA DEL DOCUMENTO HTML -->
    </xsl:template>
</xsl:stylesheet>
```

Plantilla de un documento de transformación XSL.

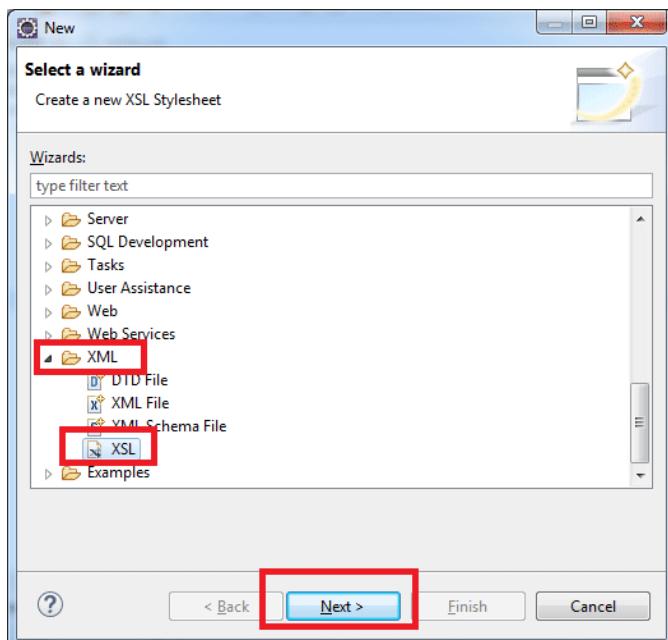
La estructura del documento HTML combina dos tipos de etiquetas:

- **Las etiquetas HTML**, tales como `<html>`, `<head>`, `<p>`, `<form>`, `<h1>`, etc.
- **Etiquetas especiales XSLT**, tales como `<xsl:value>` o `<xsl:for-each>`.

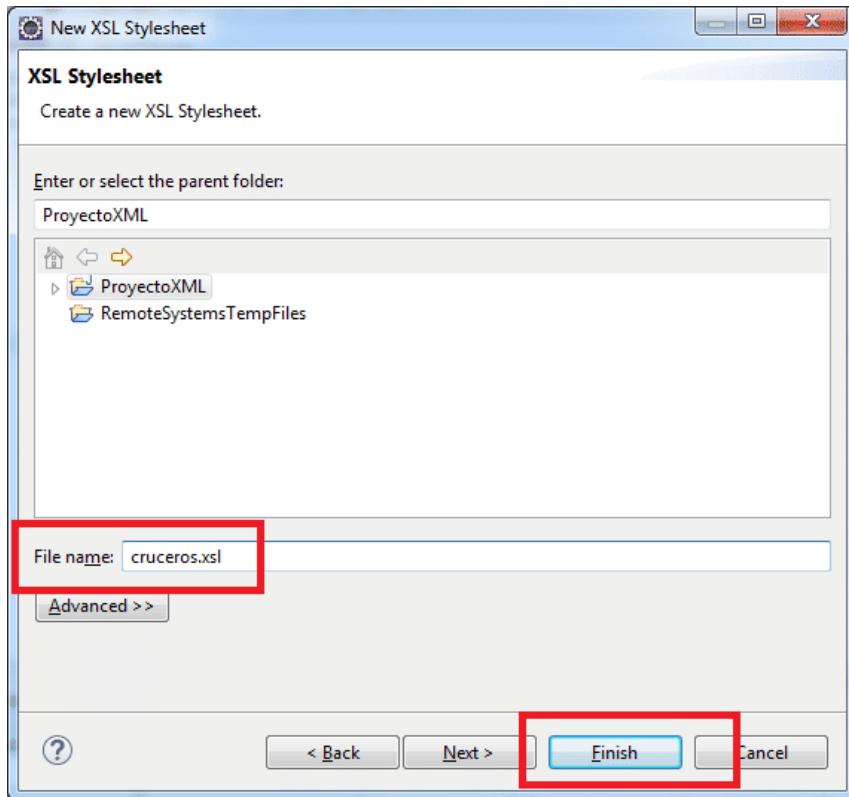
Ahora vas a crear tu primer archivo de transformación XSL y enlazarlo con el documento XML para dotarlo de estilo.

1. Crea el archivo *cruceros.xsl* para formatear el documento *cruceros.xml*

Haz clic derecho en el nombre del proyecto y selecciona en el menú contextual **New / Other**.



En el cuadro de diálogo **New**, abre la carpeta **XML**, selecciona la opción **XSL** y pulsa el botón **"Next"**.



Escribe **cruceros.xsl** como nombre de archivo y pulsa el botón **Finish**. Ya tienes la estructura de tu archivo de transformación XSL.

The screenshot shows an IDE interface with two tabs: 'cruceros.xml' and 'cruceros.xsl'. The 'cruceros.xsl' tab is active and displays the following XML code:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 @<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3 @    <xsl:template match="/">
4        <!-- TODO: Auto-generated template -->
5    </xsl:template>
6 </xsl:stylesheet>
```

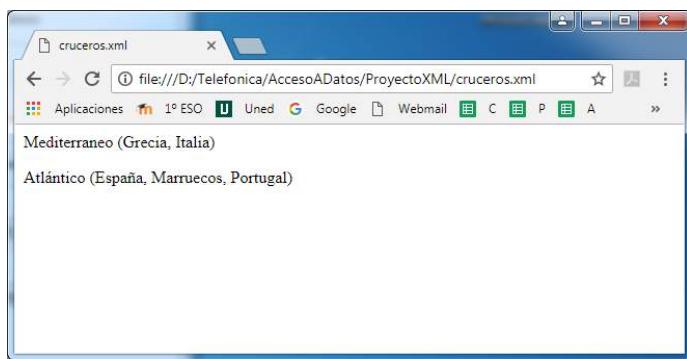
2. Edita el archivo **cruceros.xsl**

Como ves, ya tienes la estructura básica del archivo xsl. Vamos ahora a comenzar por un sencillo ejemplo que muestre el destino de los cruceros.

Edita el archivo cruceros.xsl hasta dejarlo como se muestra a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:for-each select="cruceros/crucero">
                    <p>
                        <xsl:value-of select="destino"/>
                    </p>
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Este es el resultado que esperamos obtener, una vez que hayamos enlazado el documento XML con el archivo de transformación XSL:



Vamos a analizar el código.

En primer lugar, dentro de las etiquetas **<xsl:template match="/"> ... </xsl:template>** está encerrada toda la estructura de la página web que queremos visualizar en el navegador.

Pero queremos iterar dentro del documento XML cada una de las etiquetas o elementos *crucero* para mostrar el *destino*. Para iterar o recorrer todos los elementos crucero, necesitamos una etiqueta de tipo **<xsl:for-each </xsl:for-each>**.

```
<xsl:for-each select="cruceros/crucero">
    <p>
        <xsl:value-of select="destino"/>
    </p>
</xsl:for-each>
```

Por cada uno de los cruceros iterados, por medio de la etiqueta **<xsl:value-of select="destino"/>**, estamos mostrando el elemento XML *destino*.

3. Enlaza el documento cruceros.xml con el archivo de transformación cruceros.xsl.

Para enlazar el documento XML con el archivo de transformación, debes añadir justo delante de la etiqueta raíz un elemento de tipo `<?xml-stylesheet type="text/xsl" href="cruceros.xsl"?>`. Edita el documento *cruceros.xml* para añadir la segunda línea que ves en este código:

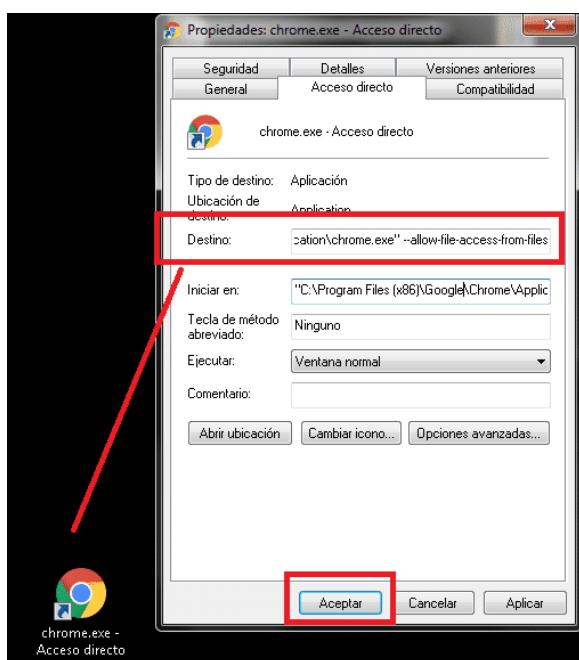
```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="cruceros.xsl"?>
<cruceros>
```

La nueva linea añadida sustituirá a la anterior, que enlazaba con el archivo CSS.

4. Visualiza el resultado en tu navegador

Abre el documento *cruceros.xml* con Internet Explorer o Mozilla Firefox. Si intentas abrirlo desde Google Chrome en modo local, no funcionará. Esto se debe a problemas de configuración de la seguridad, ya que no permite abrir el archivo xsl en local. Sin embargo, sí funcionará cuando esté subido a un *hosting web* y accedas a *cruceros.xml* en modo remoto. Podría funcionar en modo local con Chrome si abres la aplicación con el parámetro `--allow-file-access-from-files`.

Para hacer la prueba, tendrías que crear un acceso directo a la aplicación *chrome.exe*, hacer clic derecho sobre el acceso directo, seleccionar *Propiedades* en el menú contextual y añadir al final del campo destino el parámetro `--allow-file-access-from-files`. Por último, pulsamos el botón *Aceptar* y podremos visualizar nuestro documento XML, siempre y cuando hayamos abierto Chrome desde el acceso directo.



Completa el código para mostrar toda la información posible de cada crucero

Edita de nuevo el archivo *cruceros.xsl* hasta dejarlo del siguiente modo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">

        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:for-each select="cruceros/crucero">
                    <p>
                        Destino: <xsl:value-of select="destino"/>
                        <br />
                        Dias: <xsl:value-of select="detalles/cia"/>
                        <br />
                        Dias: <xsl:value-of select="detalles/dias"/>
                        <br />
                        Fecha salida: <xsl:value-of
select="detalles/fechaSalida"/>
                    </p>
                    <table>
                        <tr>
                            <th>Dia</th>
                            <th>Parada</th>
                            <th>Llegada</th>
                            <th>Salida</th>
                        </tr>
                        <xsl:for-each select="escalas/escala">
                            <tr>
                                <td><xsl:value-of select="@dia"/></td>
                                <td><xsl:value-of select="parada"/></td>
                                <td><xsl:value-of select="llegada"/></td>
                                <td><xsl:value-of select="salida"/></td>
                            </tr>
                        </xsl:for-each>
                    </table>
                </xsl:for-each>
            </body>
        </html>

    </xsl:template>
</xsl:stylesheet>
```

Ahora, el resultado visualizado en el navegador quedará así:



Vamos a analizar el nuevo código

```
<xsl:for-each select="cruceros/crucero">
    <p>
        Destino: <xsl:value-of select="destino"/>
        <br />
        Dias: <xsl:value-of select="detalles/cia"/>
        .....
    </p>
    <table>
        .....
        <xsl:for-each select="escalas/escala">
            <tr>
                <td><xsl:value-of select="@dia"/></td>
                <td><xsl:value-of select="parada"/></td>
                <td><xsl:value-of select="llegada"/></td>
                <td><xsl:value-of select="salida"/></td>
            </tr>
        </xsl:for-each>
    </table>
</xsl:for-each>
```

En primer lugar, utilizamos la siguiente estructura:

```
<xsl:for-each select="cruceros/crucero">  
.....  
</xsl:for-each>
```

Esto representa una estructura de control repetitiva que se ejecuta para cada elemento *crucero* dentro de otro elemento *cruceros*. En cada repetición se establece un filtro de modo, que delimita el contenido XML al elemento *crucero* que toque en cada iteración.

Destino: <xsl:value-of select="destino"/>
Días: <xsl:value-of select="detalles/cia"/>

Estas etiquetas permiten mostrar el *destino* y *cia* que corresponde al crucero de la iteración actual. Observa que en el atributo *select* se especifica el elemento a mostrar como referencia relativa a partir de *cruceros/crucero* (*select* especificado en la estructura *for-each*).

```
<xsl:for-each select="cruceros/crucero">  
  <xsl:for-each select="escalas/escala">  
    <tr>  
      <td><xsl:value-of select="@dia"/></td>  
      <td><xsl:value-of select="parada"/></td>  
      <td><xsl:value-of select="llegada"/></td>  
      <td><xsl:value-of select="salida"/></td>  
    </tr>  
  </xsl:for-each>  
</xsl:for-each>
```

El *for-each* interno está estableciendo filtro sobre filtro. Por cada elemento *crucero* dentro de *cruceros*, itera sus elementos *escala* dentro de *escalas* para mostrar los detalles de cada escala. El carácter *@* colocado dentro del atributo *select* identifica a un atributo de una etiqueta, no a una etiqueta, es decir, hace referencia al atributo *dia* de la etiqueta *escala* (<escala dia="1">).

Aún podríamos mejorar el aspecto estético del documento, enlazándolo con un archivo de hoja de estilos CSS. ¿Quieres aprender cómo se hace?

Usar XML, XSL y CSS

Hemos transformado el contenido XML para representarlo en el navegador dentro de una estructura HTML, pero todavía podríamos darle un estilo mucho más estético.

Para lograrlo, vamos a aplicar también CSS.

Pero, ¿qué nos aporta XSL? ¿No podríamos usar sólo CSS, como en el primer ejemplo?

XSL nos brinda la posibilidad de filtrar la información contenida en el archivo XML, algo que no es posible sólo con CSS.

Lo verás con más claridad en el apartado siguiente (*XPath*).

Sabemos que un archivo de transformación XSL encierra dentro de una plantilla la estructura de una página HTML con sus típicas etiquetas `<html>`, `<head>`, `<body>`, etc. Para utilizar también una hoja de estilo CSS, lo único que tenemos que hacer es usar una etiqueta de tipo `<link rel="stylesheet" href="css/estilo.css" />` dentro de dicha página.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head>
                <title>Cruceros</title>
                <link rel="stylesheet" href="css/estilo.css" />
            </head>
            <body>
                .....
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

En el código anterior, hemos incluido la siguiente línea dentro de la cabecera de la página:

`<link rel="stylesheet" href="css/estilo.css" />`

Ahora, sólo tienes que construir el archivo *estilo.css* dentro de una carpeta *css* con los atributos que deseas.

```
@CHARSET "ISO-8859-1";
body {
    font-family: Verdana;
}
p {
    color: blue;
}
td {
    width: 100px;
    text-align: center;
    border: solid;
}
table {
    border-collapse: collapse;
}
td.masAncho {
    width: 200px;
    text-align: left;
}
```

Además de aplicar atributos a las etiquetas HTML *body*, *p*, *td* y *table*, hemos creado una clase CSS llamada *masAncho*, aplicable a elementos de tipo *td* (celda de tabla). Eso te obliga a editar de nuevo el archivo de transformación XSL para modificar una simple línea:

```
<td class="masAncho"><xsl:value-of select="parada"/></td>
```

Si has seguido bien todos los pasos, el resultado en el navegador será el siguiente:

Día	Parada	Llegada	Salida
1	Venecia		18:00
2	Navegación		
3	Agostini	7:00	14:00
4	Santorini	9:00	20:00
5	Bari	8:00	14:00
6	Venecia	8:30	

Día	Parada	Llegada	Salida
1	Barcelona		18:00
2	Navegación		
3	Casablanca	7:00	22:00
4	Navegación		
5	Santa Cruz de Tenerife	9:00	16:00
6	Funchal	8:00	19:00
7	Barcelona	17:00	

Filtrar los resultados con XPath

XPath forma parte de la sintaxis de los documentos XSL, y tiene como objetivo buscar y seleccionar partes de un documento XML dentro de su estructura jerárquica.

En nuestro archivo de transformación XSL utilizamos la siguiente línea para iterar los distintos elementos *<crucero>* dentro de la estructura del documento XML:

```
<xsl:for-each select="cruceros/crucero">
```

Dentro del atributo *select* estás seleccionando todos los elementos *crucero* que están dentro de una etiqueta padre llamada *cruceros*. Sin saberlo, ya estás aplicando la tecnología **XPAth**.

<xsl:for-each select="cruceros/crucero">



Esto es XPAth

XPath se aplica en el valor de los atributos *select* de las etiquetas XSL.

Hasta ahora hemos utilizado la versión más simple de XPath, pero su sintaxis cuenta con caracteres especiales que brindan un sistema muy potente para la búsqueda de datos en documentos XML. Vamos a ver varios ejemplos:

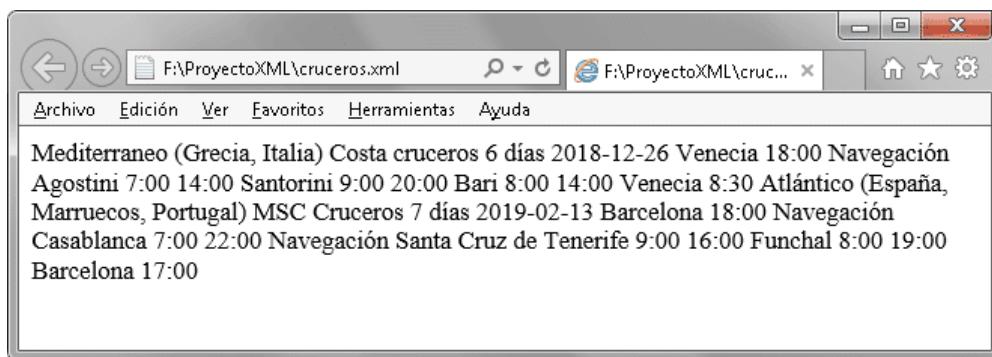
1. El carácter \ selecciona todo el contenido del elemento raíz.

Deja tu archivo de transformación XSL así de sencillo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">

        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:value-of select="\ " />
            </body>
        </html>

    </xsl:template>
</xsl:stylesheet>
```



Como resultado ofrece todo el texto contenido dentro de la etiqueta raíz, que, para nuestro documento, es la etiqueta `<cruceros>`.

2. Los caracteres // seleccionan todos los elementos situados en cualquier nivel.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">

        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:for-each select="//escala">
                    <xsl:value-of select="." /><br />
                </xsl:for-each>
            </body>
        </html>

    </xsl:template>
</xsl:stylesheet>
```

Seleccionaría todos los elementos *escala*, independientemente de la posición en la que se encuentren dentro del árbol jerárquico del documento XML.

En nuestro documento XML, todos los elementos *escala* están al mismo nivel, con lo que no notarás la diferencia si pones *cruceros/crucero/escalas/escala*. Puesto que existen varios elementos *escala*, es necesario iterar con una estructura *for-each*. En el *select* interno, el punto que aparece hace referencia al contenido de cada elemento iterado.



Resultado.

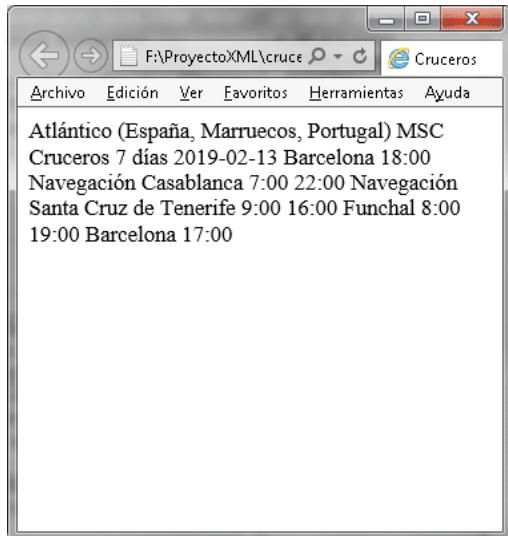
3. Podemos especificar entre corchetes [] el elemento que queremos seleccionar dentro de una colección de ellos.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">

        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:value-of select="cruceros/crucero[2]" /><br />
            </body>
        </html>

    </xsl:template>
</xsl:stylesheet>
```

Selecciona y muestra sólo el segundo crucero.



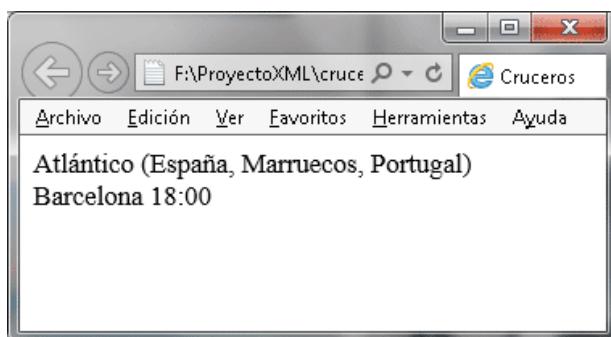
Resultado.

4. Ahora queremos seleccionar el destino del segundo crucero y el puerto de donde sale, es decir, la primera escala del segundo crucero.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">

        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:value-of select="cruceros/crucero[2]/destino" /><br />
                <xsl:value-of select="cruceros/crucero[2]/escalas/escala[1]" /><br />
            </body>
        </html>

    </xsl:template>
</xsl:stylesheet>
```



Resultado.

5. También podemos utilizar la función *last()* dentro de los corchetes.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">

        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:value-of select="cruceros/crucero[last()]/destino" /><br />
                <xsl:value-of select="cruceros/crucero[last()]/escalas/escala[1]" /><br />
            </body>
        </html>

    </xsl:template>
</xsl:stylesheet>
```

Ofrece el mismo resultado que el ejemplo anterior. Muestra el destino del último crucero, que en nuestro caso es el segundo, y luego muestra la primera escala.

6. También podemos establecer criterios de búsqueda utilizando los atributos de las etiquetas XML. En nuestro documento tenemos los atributos *codigo* de crucero y *dia* en las escalas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">

        <html>
            <head>
                <title>Cruceros</title>
            </head>
            <body>
                <xsl:value-of
select="cruceros/crucero[@codigo='CRUATL22']/destino" /><br />
                <xsl:value-of
select="cruceros/crucero[@codigo='CRUATL22']/escalas/escala[1]" /><br />
            </body>
        </html>

    </xsl:template>
</xsl:stylesheet>
```

Este ejemplo ofrece el mismo resultado que el anterior, pero filtra los cruceros por el atributo *codigo*.

¿Quieres saber más sobre XPath?

Puedes consultar el tutorial de XPath de W3Schools haciendo clic en el botón de la derecha.
https://www.w3schools.com/xml/xpath_intro.asp

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Los ficheros **XML (Extended Markup Language)** proveen un formato ligero para el intercambio de datos, lo que resulta especialmente interesante en aplicaciones web.
- **HTML se basa en el estándar SGML** que sigue un esquema jerárquico compuesto por etiquetas con apertura y cierre que tienen la siguiente estructura: `<etiqueta> contenido </etiqueta>`. Cada etiqueta en un documento XML representa un nodo o elemento, que puede estar compuesto por otros nodos.
- Es posible formatear la presentación de los documentos XML en el navegador web por medio de **hojas de estilo CSS**. Podemos aplicar atributos de estilo a las distintas etiquetas, pero no es posible aplicar filtros.
- La técnica denominada **Trasformaciones XSL (EXtensible Stylesheet Language)** está formada por un conjunto de tres lenguajes que juntos son capaces de trasformar la información contenida en los archivos XML para que pueda ser presentada al usuario en otros formatos, por ejemplo, HTML o PDF. Estos tres sublenguajes son **XSLT, XSL-FO y XPath**.
- Usando un archivo de transformación XSL podemos **establecer filtros dentro del fichero XML** para seleccionar exactamente la información que queremos presentar al usuario.

4.2. XML y JAVA (parsers)



Índice

Objetivos	3
Concepto	4
¿Qué es un parser?	4
El Parser DOM	5
Obtener el árbol DOM.....	5
Iterando los cruceros.....	7
Escribir documentos XML	11
Despedida	15
Resumen.....	15

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Comprender el concepto de *parser*.
- Construir un modelo de objeto Java a partir de un documento XML con ayuda del *parser DOM*.
- Crear un archivo XML a partir de un modelo de objetos Java con ayuda del *parserDOM*.

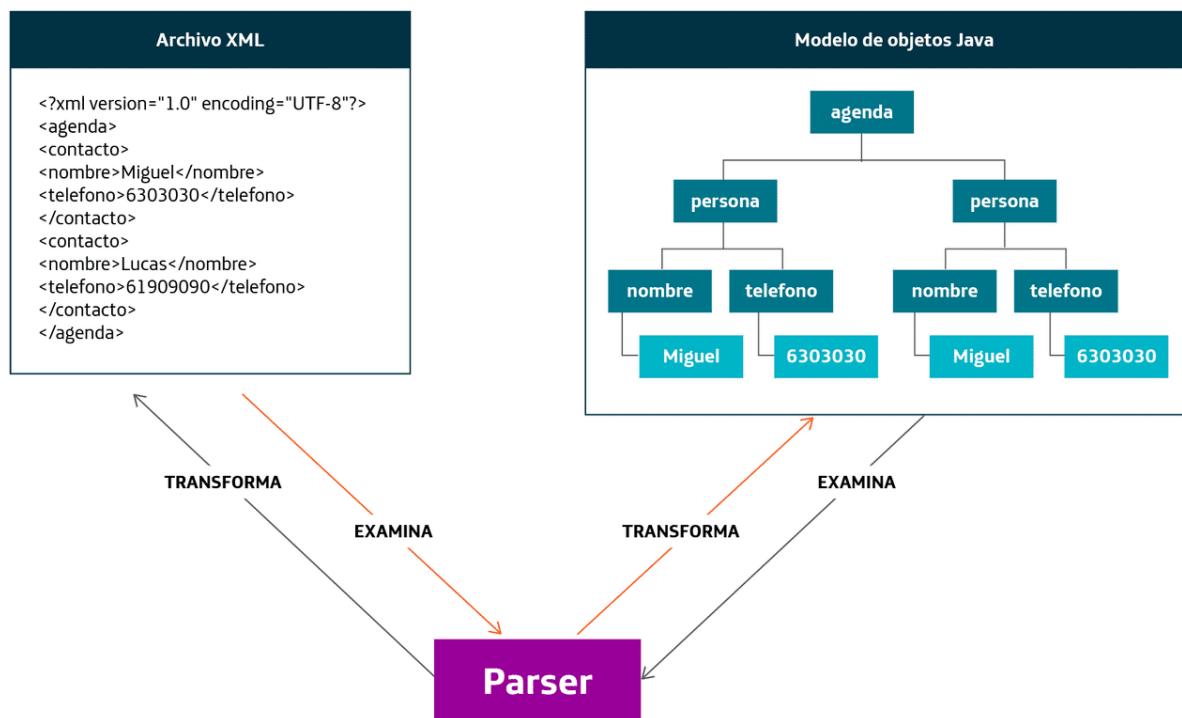
Concepto

¿Qué es un parser?

Un **parser o analizador sintáctico** es una herramienta software capaz de analizar el contenido de un documento XML y generar, a partir de él, un modelo de objetos Java. También puede realizar la operación inversa, es decir, construir a partir de un modelo de objetos Java un documento XML.

Un *parser* realiza **dos tipos de tareas**:

- Primero, a partir de un documento XML, lo examina para ver si su sintaxis es correcta. Después, una vez analizada la sintaxis, construye a partir de él un modelo de objetos que podrá ser manipulado por un programa.
- A partir de un modelo de objetos Java, lo examina con el fin de construir a partir de él un documento XML o editar uno existente.



Modelo de funcionamiento de un *parser*.

En esta lección utilizaremos el parser DOM (*Document Object Model*) que construye un modelo de objetos Java que replica la estructura del documento XML.

El Parser DOM

Obtener el árbol DOM

En este apartado utilizaremos el *parser DOM* para leer el documento `cruceros.xml` y construir, a partir de él, un árbol jerárquico de objetos Java denominado **árbol DOM** (*Document Object Model*).

Recorreremos el árbol de objetos obtenido a través del *parser DOM* para mostrar información al usuario sobre cada uno de los cruceros.

Para lograr el objetivo necesitaremos dos librerías distintas:

- **`javax.xml.parsers`:** provee clases que permiten el procesamiento de documentos XML.
- **`org.w3c.dom`:** proporciona las interfaces para la representación del DOM (*Document Object Model*).

Comenzaremos por un ejemplo simple que muestra todo el contenido de texto de la etiqueta raíz (`cruceros`), es decir, sin incluir las etiquetas, sólo los textos.

Para ponerlo en práctica, puedes abrir el proyecto de la lección anterior denominado *ProyectoXML* y añadir la clase Java siguiente:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Node;

public class LeerCruceros {
    public static void main(String[] args) {
        DocumentBuilderFactory fabrica =
DocumentBuilderFactory.newInstance();
        DocumentBuilder analizador;
        Document doc;
        Node raiz;

        try {
            analizador = fabrica.newDocumentBuilder();
            doc = analizador.parse("cruceros.xml");
            raiz = doc.getDocumentElement();
            System.out.println(raiz.getTextContent());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Vamos a analizar detenidamente el ejemplo:

DocumentBuilderFactory fabrica = DocumentBuilderFactory.newInstance();

La clase **DocumentBuilderFactory**, situada en el paquete **javax.xml.parsers**, nos permite obtener el objeto **DocumentBuilder** a partir de su método **newInstance()**. El objeto **DocumentBuilder** es imprescindible para analizar un documento XML y construir a partir de él un árbol DOM.

DocumentBuilder analizador = fabrica.newDocumentBuilder();

La clase **DocumentBuilder**, situada en el paquete **javax.xml.parsers**, representa un analizador o **parser** cuyos objetos nos permiten construir el árbol DOM a partir del documento XML por medio de su método **parse()**.

Document doc = analizador.parse("cruceros.xml");

La clase **Document**, situada en el paquete **org.w3c.dom.Document**, representa un modelo de objetos como replica de un documento XML. Es tarea del objeto **DocumentBuilder** analizar el contenido del documento XML y devolver el objeto **Document** con el árbol DOM. El objeto que en nuestro ejemplo hemos denominado *doc* ya contiene toda la estructura del documento XML.

Node raiz = doc.getDocumentElement();

Un documento XML está formado por nodos o elementos que pueden, a su vez, contener otros nodos. El método **getDocumentElement()** de la clase **Document** devuelve el objeto **Node** que representa el nodo raíz, que para nuestro ejemplo es el nodo *cruceros*.

System.out.println(raiz.getTextContent());

Nuestra variable *raiz* es una referencia al objeto **Node** que representa el nodo *cruceros*. El método **getTextContent()** muestra todo el contenido de texto sin incluir las etiquetas.

Al ejecutar el programa, la información de los cruceros se muestra más o menos así:

```

Problems @ Javadoc Declaration Console
<terminated> LeerCruceros [Java Application] C:\Program Files\Java\jdk1.8.0_6
Mediterraneo (Grecia, Italia)
Costa cruceros
6 días
2018-12-26

Venezia
18:00

Navegación

Agostini
7:00
14:00

Santorini
9:00
20:00

Bari
8:00
14:00

```

En el siguiente apartado mejoraremos la **salida a pantalla** de la información de los cruceros.

Iterando los cruceros

Ya tenemos el árbol DOM del documento *cruceros.xml*, y, a partir de él, recuperamos el nodo raíz, es decir, el nodo *cruceros*.

Ahora, mejoraremos la salida en pantalla, iterando los cruceros y mostrando de cada uno los datos que nos interesen.

En el ejemplo anterior utilizamos la expresión *raiz = doc.getDocumentElement()* para obtener el nodo raíz (*cruceros*) y a continuación mostramos, sin más, el texto contenido dentro de dicho nodo.

Sabemos que la variable *raiz* representa al nodo *cruceros*, que está compuesto por nodos hijo (etiquetas *crucero*). En esta ocasión, vamos a iterar los nodos hijos (*crucero*) y, por cada nodo *crucero*, accederemos de momento al *destino* y los *detalles*.

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class LeerCruceros {
    public static void main(String[] args) {
        DocumentBuilderFactory fabrica = DocumentBuilderFactory.newInstance();
        DocumentBuilder analizador;
        Document doc;
        Node raiz;

        try {
            analizador = fabrica.newDocumentBuilder();
            doc = analizador.parse("cruceros.xml");
            raiz = doc.getDocumentElement();
            recorrerNodos(raiz);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    private static void recorrerNodos(Node raiz) {
        NodeList nodos = raiz.getChildNodes();
        System.out.println("Elementos en el nodo raíz: " + nodos.getLength());
        for (int i=0; i<nodos.getLength();i++) {
            // Iteración por los elementos crucero.
            Node nodoHijo = nodos.item(i);
            if (nodoHijo.getNodeType() == Node.ELEMENT_NODE) {
                Node destino = nodoHijo.getChildNodes().item(1);
                System.out.println(" Destino: " + destino.getTextContent());
                Node detalles = nodoHijo.getChildNodes().item(3);
                System.out.println(" Detalles: " + detalles.getTextContent());
            }
        }
    }
}
```

Iteración de los nodos hijo (*crucero*) y acceso a *destino* y *detalles* de cada uno.

El resultado del programa queda así:

```
<terminated> LeerCruceros [Java Application] C:\Program Files\Java\jdk1.8.0_65\bin\
Elementos en el nodo raíz: 5
Destino: Mediterraneo (Grecia, Italia)
Detalles:
    Costa cruceros
    6 días
    2018-12-26

Destino: Atlántico (España, Marruecos, Portugal)
Detalles:
    MSC Cruceros
    7 días
    2019-02-13
```

Resultado.

Si observas el código, verás que nada más obtener el nodo raíz, invocamos al método `recorrerNodos()`, donde realizamos el resto de la tarea.

Céntrate en estas dos líneas:

```
NodeList nodos = raiz.getChildNodes();
System.out.println("Elementos en el nodo raíz: " + nodos.getLength());
```

Primero, obtenemos un objeto de tipo `NodeList` con la colección de nodos hijos, y después utilizamos el método `getLength()` para obtener el número total de nodos de dicha colección.

Pero, ¿por qué aparece en el resultado que hay cinco elementos, cuando sabemos que sólo hay dos cruceros?

La respuesta está en que un documento XML, además de etiquetas, contiene textos, y también admite que entre el cierre de una etiqueta y la apertura de la siguiente exista texto. Estos textos también son elementos o nodos.

En teoría, nuestro documento XML no contiene textos entre el cierre de una etiqueta y la apertura de la siguiente, pero sí contiene un retorno de carro, y eso o un simple espacio ya es considerado un texto.

Con esta imagen, que representa un documento XML muy simple, lo comprenderás mejor:

<agenda>	
Nodo de texto 1	Nodo 1: TEXT_NODE
<contacto>	
primer contacto	
<nombre>Miguel</nombre>	
<telefono>6303030</telefono>	
</contacto>	
Nodo de texto 2	Nodo 2: ELEMENT_NODE
<contacto>	
Segundo contacto	
<nombre>Lucas</nombre>	
<telefono>61909090</telefono>	
</contacto>	
Nodo de texto 3	Nodo 3: TEXT_NODE
</agenda>	Nodo 4: ELEMENT_NODE
	Nodo 5: TEXT_NODE

Documento XML simple.

Tenemos un nodo raíz *agenda* y sólo dos contactos, sin embargo, hemos metido textos entre medias de las etiquetas, con color rojo en la imagen. Estos textos también son considerados nodos. Si no estuvieran estos textos, bastaría con que hubiera un retorno de carro o un espacio en blanco en su lugar para que se considere un texto.

La clase *Node* cuenta con un método denominado *getNodeType()* que informa, mediante un número entero, de qué tipo de nodo se trata (*ELEMENT_NODE* o *TEXT_NODE*).

Vamos a terminar de analizar **el resto de código**:

```
for (int i=0; i<nodos.getLength();i++) {
    // Iteración por los elementos crucero.
    Node nodoHijo = nodos.item(i);
    if (nodoHijo.getNodeType() == Node.ELEMENT_NODE) {
        Node destino = nodoHijo.getChildNodes().item(1);
        System.out.println(" Destino: " + destino.getTextContent());
        Node detalles = nodoHijo.getChildNodes().item(3);
        System.out.println(" Detalles: " + detalles.getTextContent());
    }
}
```

Recuerda que la variable *nodos* representaba a la colección de nodos hijo dentro del nodo raíz, es decir, los elementos *crucero* y los elementos de tipo texto que no nos interesan.

Lo primero que hacemos es una iteración de tipo *for* para acceder cada uno de los nodos a través del método *item(index)*, que devuelve un elemento a través de su índice. Por cada nodo hijo iterado, sólo realizaremos el resto de las acciones si se trata de un *ELEMENT_NODE*, en cuyo caso sabemos que se trata de un elemento *crucero* con sus etiquetas internas *destino* y *detalles* que ocuparan las posiciones 1 y 3, ya que también tienen elementos de tipo texto entre medias. Ahora, **completaremos el programa** para que también muestre las escalas de cada crucero.

Ya conoces la dinámica de trabajo; ahora tendrás que considerar que cada *crucero*, además de *destino* y *detalles*, consta de un elemento *escalas* compuesto por una colección de elementos *escala*, por lo que habrá que iterar de nuevo para obtener la información de cada una de las escalas:

```

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
public class LeerCruceros {
    public static void main(String[] args) {
        DocumentBuilderFactory fabrica = DocumentBuilderFactory.newInstance();
        DocumentBuilder analizador;
        Document doc;
        Node raiz;
        try {
            analizador = fabrica.newDocumentBuilder();
            doc = analizador.parse("cruceros.xml");
            raiz = doc.getDocumentElement();
            recorrerNodos(raiz);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    private static void recorrerNodos(Node raiz) {
        NodeList nodos = raiz.getChildNodes();
        for (int i=0; i<nodos.getLength();i++) {
            // Iteración por los elementos crucero.
            Node nodoHijo = nodos.item(i);
            if (nodoHijo.getNodeType() == Node.ELEMENT_NODE) {
                Node destino = nodoHijo.getChildNodes().item(1);
                System.out.println("Destino: " + destino.getTextContent());
                System.out.println("-----");
                Node detalles = nodoHijo.getChildNodes().item(3);
                System.out.println(" Detalles: " + detalles.getTextContent());
                Node escalas = nodoHijo.getChildNodes().item(5);
                recorrerEscalas(escalas);
            }
        }
    }
    private static void recorrerEscalas(Node escalas) {
        NodeList nodos = escalas.getChildNodes();
        System.out.println(" Escalas:");
        for (int i=0; i<nodos.getLength();i++) {
            Node escala = nodos.item(i);
            if (escala.getNodeType() == Node.ELEMENT_NODE) {
                String dia = escala.getAttributes().item(0).getNodeValue();
                Node parada = escala.getChildNodes().item(1);
                Node llegada = escala.getChildNodes().item(3);
                Node salida = escala.getChildNodes().item(5);
                System.out.print(" " + dia + " : ");
                System.out.print(parada.getTextContent() + " ");
                System.out.print(llegada.getTextContent() + " - ");
                System.out.println(salida.getTextContent());
            }
        }
        System.out.println();
    }
}

```

Iteración de elementos *escala*.

Nuestro programa arroja el siguiente **resultado**:

The screenshot shows a Java application window with tabs: Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a program named 'LeerCruceros'. The output includes:

- Destino: Mediterraneo (Grecia, Italia)
 - Detalles:
 - Costa cruceros
 - 6 días
 - 2018-12-26
 - Escalas:
 - Día 1: Venecia - 18:00
 - Día 2: Navegación -
 - Día 3: Agostini 7:00 - 14:00
 - Día 4: Santorini 9:00 - 20:00
 - Día 5: Bari 8:00 - 14:00
 - Día 6: Venecia 8:30 -
- Destino: Atlántico (España, Marruecos, Portugal)
 - Detalles:
 - MSC Cruceros
 - 7 días
 - 2019-02-13
 - Escalas:
 - Día 1: Barcelona - 18:00
 - Día 2: Navegación -
 - Día 3: Casablanca 7:00 - 22:00
 - Día 4: Navegación -
 - Día 5: Santa Cruz de Tenerife 9:00 - 16:00
 - Día 6: Funchal 8:00 - 19:00
 - Día 7: Barcelona 17:00 -

Vista del resultado final.

Escribir documentos XML

La tecnología DOM también nos permite construir archivos XML a partir de un modelo de objetos Java.

Aprenderás cómo a partir de un sencillo ejemplo, que crea un documento XML para guardar los datos de una agenda telefónica.

Comienza por crear un proyecto Java y la siguiente clase principal, luego analizaremos el código detenidamente.

```
import java.io.File;  
  
import javax.xml.parsers.DocumentBuilder;  
import javax.xml.parsers.DocumentBuilderFactory;  
import javax.xml.transform.Transformer;  
import javax.xml.transform.TransformerException;  
import javax.xml.transform.TransformerFactory;  
import javax.xml.transform.dom.DOMSource;  
import javax.xml.transform.stream.StreamResult;  
  
import org.w3c.dom.Document;  
import org.w3c.dom.Element;
```

```
public class CrearAgenda {  
    public static void main(String[] args) {  
        DocumentBuilderFactory fabrica = DocumentBuilderFactory.newInstance();  
        DocumentBuilder analizador;  
        Document doc;  
  
        try {  
            analizador = fabrica.newDocumentBuilder();  
            // Creamos nuevo documento  
            doc = analizador.newDocument();  
            // Añadimos elemento raiz  
            Element agenda = doc.createElement("agenda");  
            doc.appendChild(agenda);  
            // Añadimos tres contactos al elemento raiz agenda.  
            agregarContactos(agenda, doc);  
            // Guardamos en disco el nuevo documento XML.  
            guardar(doc);  
  
            System.out.println("El archivo se ha creado con éxito");  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
  
    public static void agregarContactos(Element agenda, Document doc) {  
        // Agregamos primer contacto  
        Element contacto = doc.createElement("contacto");  
        agenda.appendChild(contacto);  
        Element nombre = doc.createElement("nombre");  
        nombre.appendChild(doc.createTextNode("AMELIA GONZALEZ LOPEZ"));  
        contacto.appendChild(nombre);  
        Element telefono = doc.createElement("telefono");  
        telefono.appendChild(doc.createTextNode("612333333"));  
        contacto.appendChild(telefono);  
  
        // Agregamos segundo contacto  
        contacto = doc.createElement("contacto");  
        agenda.appendChild(contacto);  
        nombre = doc.createElement("nombre");  
        nombre.appendChild(doc.createTextNode("PEDRO BOTIJO MONTERA"));  
        contacto.appendChild(nombre);  
        telefono = doc.createElement("telefono");  
        telefono.appendChild(doc.createTextNode("622333444"));  
        contacto.appendChild(telefono);  
  
        // Agregamos tercer contacto  
        contacto = doc.createElement("contacto");  
        agenda.appendChild(contacto);  
        nombre = doc.createElement("nombre");  
        nombre.appendChild(doc.createTextNode("MIGUEL MALATESTA SENTADO"));  
        contacto.appendChild(nombre);  
        telefono = doc.createElement("telefono");  
        telefono.appendChild(doc.createTextNode("655444333"));  
        contacto.appendChild(telefono);  
    }  
  
    private static void guardar(Document doc) throws TransformerException {  
        TransformerFactory fabricaConversor = TransformerFactory.newInstance();  
        Transformer conversor = fabricaConversor.newTransformer();  
        DOMSource fuente = new DOMSource(doc);  
        StreamResult resultado = new StreamResult(new File("agenda.xml"));  
        conversor.transform(fuente, resultado);  
    }  
}
```

Creación del proyecto y la clase principal.

Vamos a estudiar detenidamente el código anterior:

doc = analizador.newDocument();

De nuevo volvemos a utilizar una referencia a un objeto de tipo *DocumentBuilder* (*analizador*) para obtener un objeto *Document*. Esta vez queremos construir un objeto *Document* nuevo para después ir añadiéndole nuevos elementos o nodos.

Element agenda = doc.createElement("agenda");

Los objetos *Document* cuentan con el método *createElement()* capaz de crear un objeto de tipo *Element*. La clase *Element* representa un elemento de un documento XML, o lo que es lo mismo, una etiqueta con su apertura y cierre. Un elemento también es un nodo, sin embargo, un objeto de la clase *Node* puede representar a cualquier tipo de nodo (elementos o textos), mientras que un objeto de tipo *Element* representa a una etiqueta.

doc.appendChild(agenda);

Con la sentencia "*Element agenda = doc.createElement("agenda")*", creamos un elemento XML virtual, y ahora con el método *appendChild(agenda)* lo estamos agregando al documento. Hasta aquí, nuestro documento virtual está así:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<agenda>
</agenda>
```

A partir de ahora, el programa deberá ir añadiendo nuevos elementos hijos dentro del elemento raíz *agenda*.

agregarContactos(agenda, doc);

Dentro de este método agregamos tres contactos, ejecutando las siguientes sentencias:

```
Element contacto = doc.createElement("contacto");
agenda.appendChild(contacto);
Element nombre = doc.createElement("nombre");
nombre.appendChild(doc.createTextNode("AMELIA GONZALEZ LOPEZ"));
contacto.appendChild(nombre);
Element telefono = doc.createElement("telefono");
telefono.appendChild(doc.createTextNode("61233333"));
contacto.appendChild(telefono);
```

Para añadir cada uno de los contactos, realizamos las siguientes operaciones:

- Creamos un elemento *contacto* y lo añadimos al elemento *agenda*.
- Creamos un elemento *nombre* y le añadimos un elemento de texto con el valor "AMELIA GONZALEZ LOPEZ".
- Añadimos el elemento *nombre* al contacto.
- Creamos un elemento *telefono* y le añadimos un elemento de texto con el valor "61233333".
- Añadimos el elemento *telefono* al contacto.

guardar(doc);

Hasta aquí, se ha creado el árbol DOM para el nuevo documento XML, pero en memoria, no se ha guardado en un archivo de disco. Justo ésa es la misión del método *guardar*. Vamos a recordar la implementación del método *guardar()*.

```
TransformerFactory fabricaConversor = TransformerFactory.newInstance();
Transformer conversor = fabricaConversor.newTransformer();
DOMSource fuente = new DOMSource(doc);
StreamResult resultado = new StreamResult(new File("agenda.xml"));
conversor.transform(fuente, resultado);
```

La clase *Transformer* dispone de métodos que transforman un árbol DOM a un formato de documento XML. Hay que tener en cuenta que la estructura DOM no es exclusiva de los documentos XML; hay otros formatos a partir de los que se puede obtener un árbol DOM, por ejemplo, un documento HTML.

Para obtener una referencia a un objeto *Transformer* hay que invocar al método *newTransformer()* de la clase *TransformerFactory*.

Este objeto *Transformer* provee del método *transform()* que realiza la transformación al formato XML, generando así el documento físico. El método *transform()* requiere dos argumentos:

- Un objeto *DOMSource*, que representa el árbol DOM de origen.
- Un objeto *StreamResult*, que representa un flujo de datos de escritura asociado al fichero físico donde deseamos escribir los datos XML.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Un **parsero analizador sintáctico** es una herramienta software capaz de analizar el contenido de un documento XML y generar a partir de él un modelo de objetos Java. También puede realizar la operación inversa, es decir, construir a partir de un modelo de objetos Java un documento XML.
- Uno de los *parsers* más empleados en las aplicaciones Java es el **parser DOM (Document Object Model)**. El parser DOM requiere estas librerías: **javax.xml.parsers**, y provee clases que permiten el procesamiento de documentos XML y **org.w3c.dom**, que proporciona las interfaces para la representación del DOM.

4.3. Bases de datos XML vs bases de datos relacionales



Índice

Objetivos	3
Bases de datos XML	4
Qué son las bases de datos XML	4
Document Type Definition (DTD)	5
XML Schema Definition (XSD)	9
Tecnologías asociadas a las bases de datos XML	14
Bases de datos documentales	15
Qué son las bases de datos documentales	15
Base de datos XML vs Base de datos relacional.....	15
Despedida	17
Resumen.....	17

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Comprender qué son las bases de datos XML y su relación con las bases de datos documentales.
- Comparar la estructura de una base de datos XML y una base de datos relacional.
- Definir la estructura de una base de datos XML y aplicar restricciones mediante dos técnicas distintas: DTD (*Document Type Definition*) y XSD (*XML Schema Definition*)

Bases de datos XML

Qué son las bases de datos XML

Una **base de datos XML** está constituida por uno o varios documentos XML, que siguen una estructura lógica similar a la estructura de filas y columnas de una base de datos tradicional, pero con varios niveles.

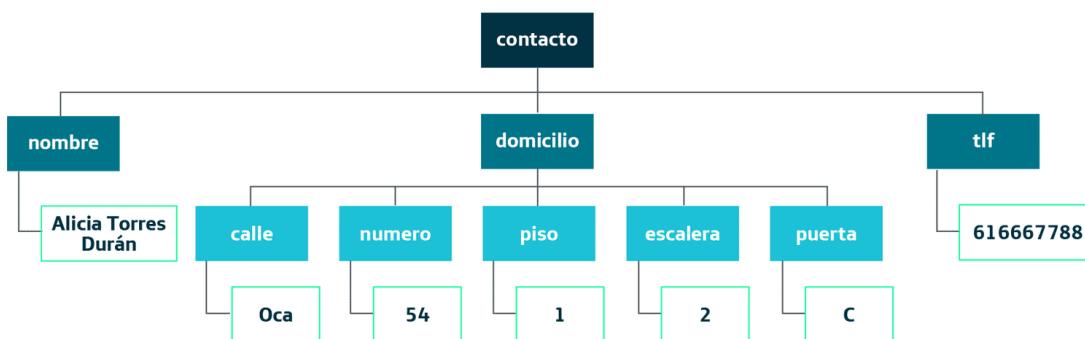
Esta estructura debe ser validada utilizando las tecnologías disponibles.

Cada archivo XML estará constituido por una etiqueta raíz, cuyo nombre debería identificar el tipo de objetos o entidades que se almacenan en él. El siguiente ejemplo refleja una agenda compuesta por objetos *contacto*. Podemos considerar que cada *contacto* es un registro.

```
<agenda>
    <contacto>
        <nombre> Alicia Torres Durán</nombre>
        <domicilio>
            <calle>Oca</calle>
            <numero>54</numero>
            <piso>1</piso>
            <escalera>2</escalera>
            <puerta>C</puerta>
        </domicilio>
        <tlf> 616667788</tlf>
    </contacto>
    .....
</agenda>
```

Ejemplo de una agenda compuesta por objetos *contacto*.

Cada registro (*contacto* en nuestro ejemplo) sigue una estructura jerárquica en varios niveles que puede representarse como en la siguiente imagen:



Estructura jerárquica en varios niveles.

Además, para que un archivo XML pueda ser considerado base de datos, o parte de una base de datos, debe cumplir unos criterios de validación. Para nuestro ejemplo, estos criterios de validación exigirían que cada uno de los contactos tenga la misma estructura, y que datos como el número, piso y escalera sean valores numéricos.

En los siguientes apartados aprenderás a **establecer criterios de validación para un documento XML**, utilizando dos técnicas distintas:

- *Document Type Definition (DTD)*.
- *XML Schema Definition (XSD)*.

Document Type Definition (DTD)

Un **DTD (Document Type Definition)** es un archivo que define la estructura de un documento XML, es decir, el nombre de los elementos que contiene, sus atributos, el orden en el que tiene que aparecer cada elemento, si se repiten, y si pueden o no tener elementos hijos.

A continuación te mostraremos cómo sería el archivo DTD para validar la estructura del documento **cruceros.xml** con el que trabajamos anteriormente.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT cruceros (crucero*)>
<!ELEMENT crucero (destino, detalles, escalas)>

<!ELEMENT destino (#PCDATA)>

<!ELEMENT detalles (cia, dias, fechaSalida)>
<!ELEMENT cia (#PCDATA)>
<!ELEMENT dias (#PCDATA)>
<!ELEMENT fechaSalida (#PCDATA)>

<!ELEMENT escalas (escala*)>
<!ELEMENT escala (parada, llegada, salida)>
<!ELEMENT parada (#PCDATA)>
<!ELEMENT llegada (#PCDATA)>
<!ELEMENT salida (#PCDATA)>

<!ATTLIST crucero codigo CDATA #REQUIRED>
<!ATTLIST escala dia CDATA #REQUIRED>
```

Vamos a analizarlo por partes:

- **<!ELEMENT cruceros (crucero*)>**
<!ELEMENT crucero (destino, detalles, escalas)>

Aquí, estamos indicando que el elemento cruceros estará compuesto por un conjunto de elementos crucero, cada uno de los cuales estará compuesto a su vez por los elementos internos destino, detalles y escalas.

- **<!ELEMENT destino (#PCDATA)>**

De este modo indicamos que el elemento o etiqueta destino estará compuesto por un dato elemental, es decir, que no se descompone en más etiquetas hijas o elementos.

- **<!ELEMENT detalles (cia, dias, fechaSalida)>**
<!ELEMENT cia (#PCDATA)>
<!ELEMENT dias (#PCDATA)>
<!ELEMENT fechaSalida (#PCDATA)>

En este caso, indicamos que el elemento detalles estará compuesto por los elementos hijos cia, dias y fechaSalida. Por otro lado, cada uno de estos tres elementos hijo son datos elementales que no se descomponen en más niveles.

- **<!ELEMENT escalas (escala*)>**
<!ELEMENT escala (parada, llegada, salida)>
<!ELEMENT parada (#PCDATA)>
<!ELEMENT llegada (#PCDATA)>
<!ELEMENT salida (#PCDATA)>

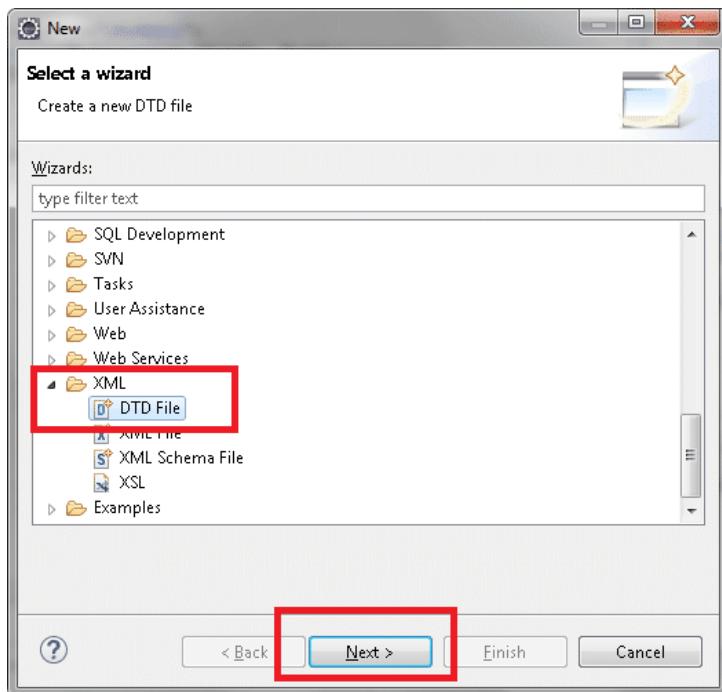
Aquí indicamos que el elemento escalas estará compuesto por un conjunto de elementos hijos escala. Cada uno de los elementos escala estará, a su vez, compuesto por los elementos hijo parada, llegada y salida de tipo elemental.

- **<!ATTLIST crucero codigo CDATA #REQUIRED>**
<!ATTLIST escala dia CDATA #REQUIRED>

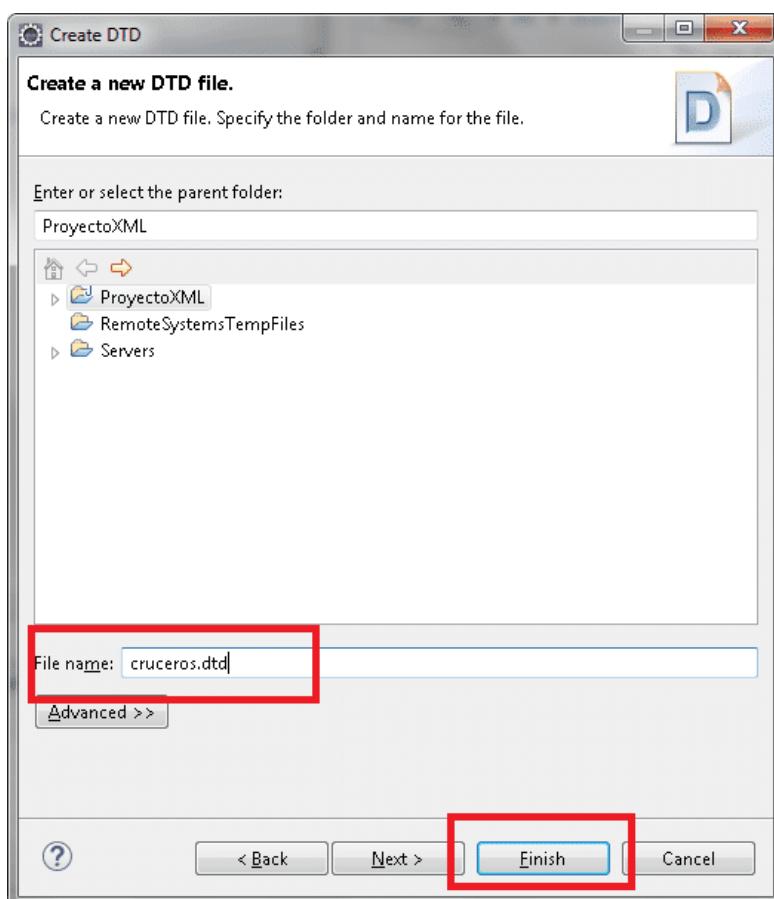
Por último, estamos definiendo los atributos codigo y dia para las etiquetas crucero y escala como requeridos.

Sigue estos pasos para crear el archivo DTD desde Eclipse:

1. Abre el proyecto Eclipse, llamado **ProyectoXML**, que creaste en la primera lección de esta unidad didáctica. Recuerda que en ese proyecto tienes el archivo *cruceros.xml* que enlazarás con el archivo DTD.
2. Haz clic derecho sobre el nombre del proyecto y selecciona **New / Other** en el menú contextual.
3. En el cuadro de diálogo **New**, abre la carpeta **XML** y selecciona la opción **DTD file**. Tendrás que utilizar la barra de desplazamiento vertical. Luego pulsa el botón **Next >**.



4. Escribe **cruceros.dtd** como nombre del nuevo fichero y pulsa el botón **Finish**.



5. **Completa el código** del archivo DTD y guarda los cambios. Puedes copiar el código de esta misma lección que aparece más arriba.

```

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT cruceros (crucero*)>
<!ELEMENT crucero (destino, detalles, escalas)>
<!ELEMENT destino (#PCDATA)>
<!ELEMENT detalles (cia, dias, fechaSalida)>
<!ELEMENT cia (#PCDATA)>
<!ELEMENT dias (#PCDATA)>
<!ELEMENT fechaSalida (#PCDATA)>
<!ELEMENT escalas (escala*)>
<!ELEMENT escala (parada, llegada, salida)>
<!ELEMENT parada (#PCDATA)>
<!ELEMENT llegada (#PCDATA)>
<!ELEMENT salida (#PCDATA)>
<!ATTLIST crucero codigo CDATA #REQUIRED>
<!ATTLIST escala dia CDATA #REQUIRED>

```

Sigue los siguientes pasos para enlazar el archivo DTD con el documento XML:

1. Abre el archivo **cruceros.xml** y añade la siguiente línea justo después de la cabecera del documento XML: `<!DOCTYPE cruceros SYSTEM "cruceros.dtd" >`

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cruceros SYSTEM "cruceros.dtd" >
<cruceros>
  <crucero codigo="CRUMED21">
    <destino>Mediterraneo (Grecia, Italia)</destino>
    <detalles>
      <cia>Costa cruceros</cia>
      <dias>6 días</dias>
      <fechaSalida>2018-12-26</fechaSalida>
    </detalles>
    <escalas>
      <escala dia="1">
        <parada>Venecia</parada>
        <llegada></llegada>
        <salida>18:00</salida>
      </escala>
    </escalas>
  </crucero>
</cruceros>

```

Con la nueva línea estamos aplicando las restricciones especificadas en el archivo *cruceros.dtd* a la etiqueta raíz *cruceros*.

Ahora, puedes comprobar que las restricciones se están aplicando, probando a editar el código del documento *cruceros.xml*. Las siguientes acciones provocarían error en el código:

- Eliminar el atributo *codigo* en alguno de los cruceros, ya que es un atributo requerido.
- Eliminar el atributo *dia* en alguna de las escalas, ya que es un atributo requerido.

- Intercambiar el orden de alguno de los atributos, por ejemplo: en *detalles* poner *cia*, *fechaSalida*, *dias*, en lugar de *cia*, *dias*, *fechaSalida* como está establecido.
- Eliminar alguna de las etiquetas en uno de los cruceros, ya que todas las etiquetas definidas en el archivo DTD deben estar presentes en cada *crucero* y en el orden establecido.

XML Schema Definition (XSD)

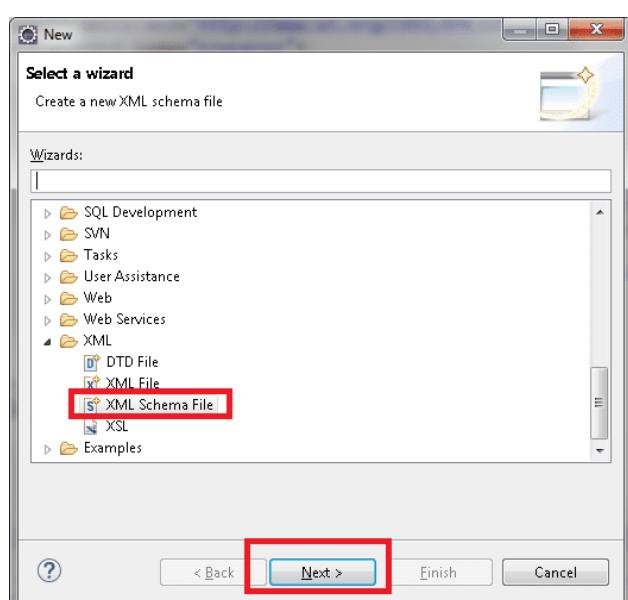
XSD es un lenguaje utilizado para definir la estructura y las restricciones de un documento XML. Fue desarrollado por el W3C (*World Wide Web Consortium*) y alcanzó popularidad a partir del año 2001, desplazando a la tecnología DTD.

Con DTD, definíamos las etiquetas que debía tener el documento *cruceros.xml* y el orden en que debían aparecer, pero no era posible especificar el tipo de dato asociado a las etiquetas elementales, es decir, no había manera de indicar que el *destino* del *crucero* es un campo de texto y los *dias* es un dato numérico. Esta limitación fue superada por los archivos XSD.

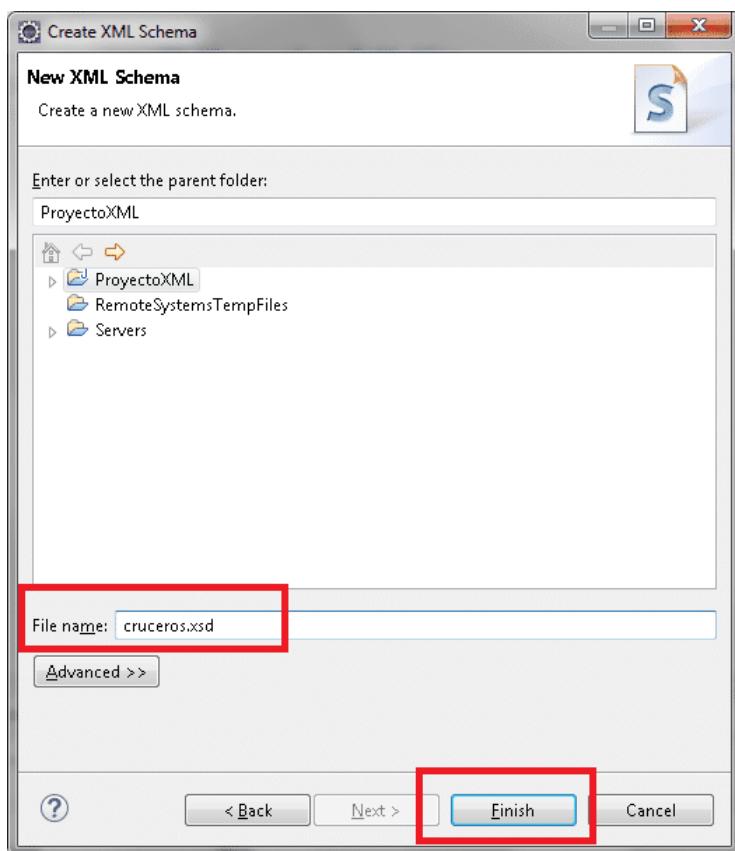
Para facilitar la comprensión de esta nueva tecnología, no vamos a desarrollar el esquema XSD completo, sino que iremos dando pasos hasta llegar a la versión completa.

Sigue estos pasos para crear la primera versión del archivo XSD en Eclipse:

1. Haz clic derecho sobre el nombre del proyecto y selecciona **New / Other** en el menú contextual.
2. Utiliza la barra de desplazamiento vertical en el cuadro de diálogo **New** si es necesario hasta encontrar la carpeta **XML**, y selecciona la opción **XML Schema File**. Luego pulsa el botón **Next >**.



3. Escribe **cruceros.xsd** como nombre de archivo y pulsa el botón **Finish**.



4. Sustituye el contenido que viene por defecto por el código siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="cruceros">
        <xsd:complexType>
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="crucero">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="destino"
type="xsd:string" />
                            <xsd:element name="detalles">
</xsd:element>
                            <xsd:element name="escalas">
</xsd:element>
                            </xsd:sequence>
                            <xsd:attribute name="codigo"
type="xsd:string" />
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
```

Vamos a analizar el código detenidamente:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    .....
</xsd:schema>
```

En primer lugar, **un archivo XSD es un documento XML con etiquetas especiales que tienen como función definir la estructura y restricciones de otro documento XML**. En nuestro caso, el archivo *cruceros.xsd* tendrá como objetivo definir la estructura del archivo *cruceros.xml*.

Los archivos XSD tienen que contar con una etiqueta principal `<xsd:schema>` que encierra el resto de etiquetas que irán configurando las distintas restricciones.

Con el atributo `xmlns:xsd` y el valor `http://www.w3.org/2001/XMLSchema` estamos indicando la URL donde se definen los tipos de datos y etiquetas especiales para los archivos XSD. También estamos indicando que debemos utilizar el prefijo `xsd` en cada una de las etiquetas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="cruceros">
        <xsd:complexType>
            </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Recuerda que se trata de definir la estructura del documento *cruceros.xml*. Pues bien, estamos indicando que debe constar, en primer lugar, una etiqueta o elemento llamado *cruceros* y que, además, es de *tipo complejo*, es decir, es una etiqueta que se subdivide en más etiquetas. Entre las etiquetas `<xsd:complexType> </xsd:complexType>` será necesario configurar la composición que debe tener la etiqueta *cruceros*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="cruceros">
        <xsd:complexType>
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="crucero">
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

En las nuevas líneas añadidas indicamos que el elemento *cruceros* está compuesto por una secuencia que va desde 0 a infinitos elementos de tipo *crucero*.

```
.....
<xsd:element name="crucero">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="destino" type="xsd:string" />
            <xsd:element name="detalles"> </xsd:element>
            <xsd:element name="escalas"> </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="codigo" type="xsd:string" />
    </xsd:complexType>
</xsd:element>
.....
```

Ahora estamos indicando que cada elemento *crucero* es de tipo complejo, es decir, que se descompone en otras etiquetas. Además indicamos que cada *crucero* está compuesto por una secuencia de tres etiquetas denominadas *destino* (elemental y de tipo *string*), *detalles* y *escalas*.

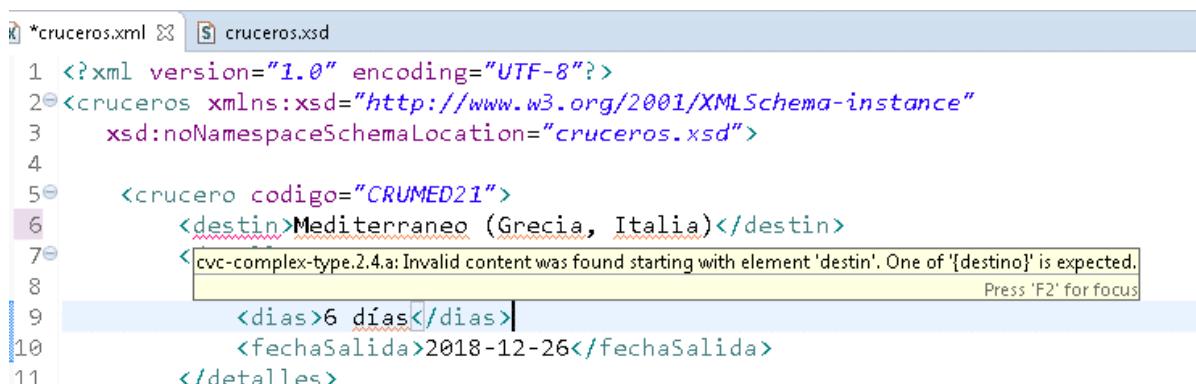
Todavía queda trabajo pendiente, ya que será necesario definir, además, la composición de las etiquetas *detalles* y *escalas*, que también son de tipo complejo. No obstante, aunque no hayamos terminado del todo, ya podemos enlazar el archivo XSD con el documento *cruceros.xml* y comenzar a ver el resultado de nuestro trabajo.

Para enlazar el archivo *cruceros.xml* con el esquema XSD que acabamos de crear, tienes que añadir unos atributos especiales a la etiqueta *cruceros*.

Para empezar, debes **editar el archivo *cruceros.xsd*** para que comience de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<cruceros xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
    xsd:noNamespaceSchemaLocation="cruceros.xsd">
```

Como prueba, puedes editar el archivo *cruceros.xsd* en Eclipse y realizar algún cambio que contradiga al archivo XSD. Por ejemplo, sustituye el nombre de la etiqueta *destino* por *destin* y comprueba cómo eclipse marca dicha etiqueta con subrayado rojo como error.



Si sitúas el puntero de razón sobre el subrayado rojo, te aparecerá la descripción del error en un recuadro amarillo.

Crea la versión final de XML SCHEMA:

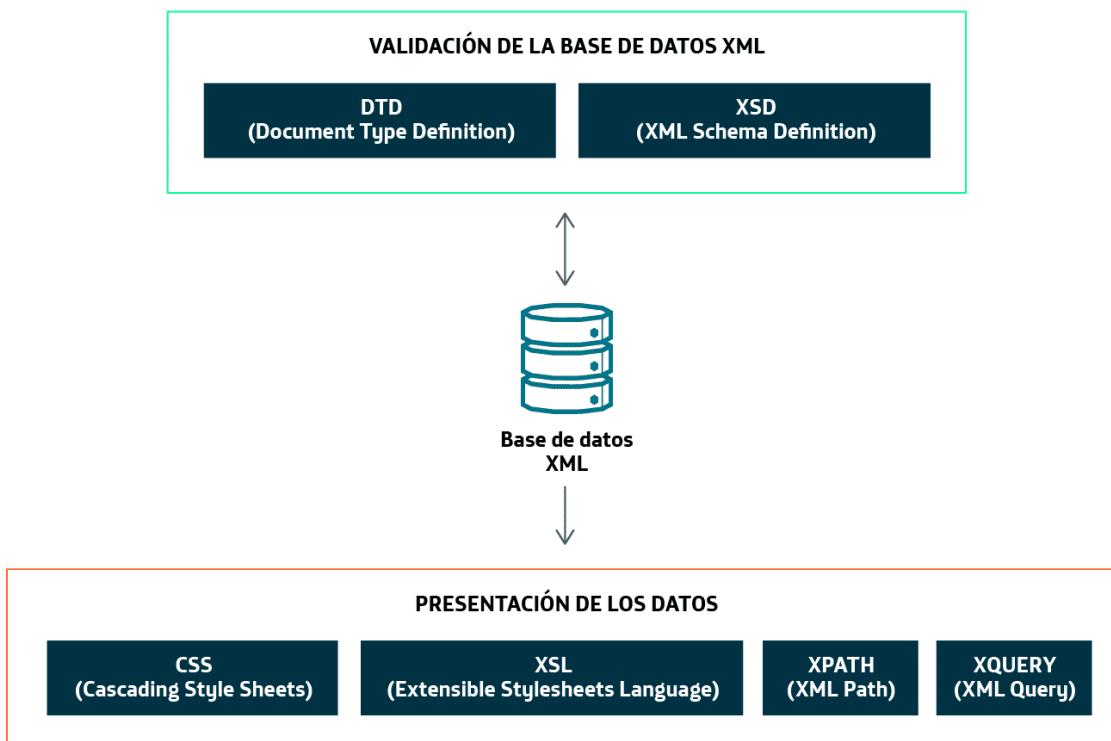
Todavía tenemos pendiente la definición de la estructura de los elementos *detalles* y *escalas*. Ambos son de tipo complejo, pero *detalles* se descompone en otros tres elementos que sólo se repiten una vez, mientras que *escalas* es una secuencia de 0 a infinitos elementos *escala*, cada uno de ellos elementos complejos compuestos por los subelementos *parada*, *llegada* y *salida*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="cruceros">
        <xsd:complexType>
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="crucero">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="destino" type="xsd:string" />
                            <xsd:element name="detalles">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element name="cia" type="xsd:string" />
                                        <xsd:element name="dias" />
                                        <xsd:element name="fechaSalida" type="xsd:date" />
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                            <xsd:element name="escalas">
                                <xsd:complexType>
                                    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                                        <xsd:element name="escala">
                                            <xsd:complexType>
                                                <xsd:sequence>
                                                    <xsd:element name="parada" type="xsd:string" />
                                                    <xsd:element name="llegada" type="xsd:string" />
                                                    <xsd:element name="salida" type="xsd:string" />
                                                </xsd:sequence>
                                                <xsd:attribute name="dia" type="xsd:int" />
                                            </xsd:complexType>
                                        </xsd:element>
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Tecnologías asociadas a las bases de datos XML

Ya conoces las tecnologías asociadas a las bases de datos XML: tecnologías para validación (DTD, XSD) y tecnologías para la presentación de los datos al usuario (CSS, XSL, XPATH, XQUERY).

A modo de resumen, la siguiente imagen esquematiza todas estas tecnologías:



Ya estudiaste en la primera lección de esta unidad las tecnologías para la presentación de los datos CSS, XSL y XPATH. Pero te falta por aprender el lenguaje para la selección de datos XQUERY, al que dedicaremos una lección completa más adelante.

Ahora que conoces las bases de datos XML, vamos a profundizar en las bases de datos documentales.

Bases de datos documentales

Qué son las bases de datos documentales

Una **base de datos documental** es un conjunto de información estructurada en registros, donde cada registro constituye una unidad autónoma de información que puede estar, a su vez, estructurada en diferentes niveles, constituyendo una estructura jerárquica o en árbol.

El concepto de base de datos XML está directamente relacionado con las **bases de datos documentales**.

En las bases de datos documentales, cada registro se corresponde con un documento completo de cualquier tipo: un albarán, una factura, la ficha de un libro, etc.

Considerando nuestro archivo *cruceros.xml* como una base de datos, cada etiqueta *crucero* constituye un registro a partir del cual podemos obtener un documento completo que describe a un crucero.

Base de datos XML vs Base de datos relacional

En este apartado vamos a **comparar la estructura de una base de datos relacional con la de una base de datos XML**.

- **Para obtener un documento en una base de datos relacional, necesitamos consultar varias relaciones o tablas y, por cada una de ellas, varios registros o filas.**
Pensemos en la impresión de una factura: para obtener dicho documento necesitamos consultar en una base de datos las tablas de clientes, artículos, facturas, detalles de facturas, etc.
- **Para obtener un documento en una base de datos XML, que corresponde con un formato documental, necesitamos consultar un único registro, que constituye una unidad autónoma.**

Una base de datos relacional que contenga la información de nuestros cruceros podría tener la siguiente estructura:

CRUCERO				
CODIGO	DESTINO	CIA	DIAS	FECHA
CRUMED21	Mediterraneo (Grecia, Italia)	Costa cruceros	6 días	2018-12-26
CRUATL22	Atlántico (España, Marruecos, Portugal)	MSC Cruceros	7 días	2019-02-13

ESCALA				
CODIGO	DIA	PARADA	LLEGADA	SALIDA
CRUMED21	1	Venecia		18:00
CRUMED21	2	Navegación		
CRUMED21	3	Agostini	7:00	14:00
CRUMED21	4	Santorini	9:00	20:00
CRUMED21	5	Bari	8:00	14:00
CRUMED21	6	Venecia	8:30	
CRUATL22	1	Barcelona		18:00
CRUATL22	2	Navegación		
CRUATL22	3	Casablanca	7:00	22:00
CRUATL22	4	Navegación		
CRUATL22	5	Sta Cruz de Tenerife	9:00	16:00
CRUATL22	6	Funchal	8:00	19:00
CRUATL22	7	Barcelona	17:00	

Ejemplo de base de datos relacional.

Para obtener un documento descriptivo de uno de los cruceros, habrá que consultar una fila de la tabla *CRUCERO* y varias filas de la tabla *ESCALA*.

Cada registro en la base de datos XML de cruceros está encerrado en una etiqueta *<crucero>*, que tiene una estructura como esta:

```

<crucero codigo="CRUMED21">
  <destino>Mediterraneo (Grecia, Italia)</destino>
  <detalles>
    <cia>Costa cruceros</cia>
    <dias>6 días</dias>
    <fechaSalida>2018-12-26</fechaSalida>
  </detalles>
  <escalas>
    <escala dia="1">
      <parada>Venecia</parada>
      <llegada></llegada>
      <salida>18:00</salida>
    </escala>
    <escala dia="2">
      <parada>Navegación</parada>
      <llegada></llegada>
      <salida></salida>
    </escala>
    .....
  </escalas>
</crucero>

```

Estructura de la etiqueta *crucero*.

Cada elemento *crucero* encierra un registro que contiene toda la información necesaria para elaborar un documento descriptivo del crucero.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Una **base de datos XML** está constituida por uno o varios documentos XML que siguen una estructura lógica similar a la estructura de filas y columnas de una base de datos tradicional, pero con varios niveles. Esta estructura debe ser validada utilizando las tecnologías disponibles a este efecto.
- Para que un archivo XML pueda ser considerado base de datos, o parte de una base de datos, debe cumplir unos **criterios de validación** que pueden establecerse con una de las siguientes técnicas: *Document Type Definition (DTD)* o *XML Schema Definition (XSD)*.
- El concepto de base de datos XML está directamente relacionado con las llamadas bases de datos documentales. Una **base de datos documental** es un conjunto de información estructurada en registros, donde cada registro constituye una unidad autónoma de información que puede estar, a su vez, estructurada en diferentes niveles, constituyendo una estructura jerárquica o en árbol.
- En las bases de datos documentales, cada registro se corresponde con un documento completo de cualquier tipo, un albarán, una factura, la ficha de un libro, etc.

4.4. Instalación y uso de BaseX como sistema de gestión de bases de datos XML



Índice

Objetivos	3
Gestor de bases de datos XML BaseX	4
¿Qué es BaseX?	4
Obtener e instalar BaseX	4
Utilizar BaseX.....	5
Despedida	6
Resumen.....	6

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Obtener el gestor de bases de datos XML BaseX de forma gratuita en Internet.
- Crear una nueva base de datos en BaseX.
- Manipular una base de datos con BaseX.

Gestor de bases de datos XML

BaseX

¿Qué es BaseX?

BaseX es un sistema de gestión de bases de datos XML que provee de dos herramientas principales.

- **Una interfaz de usuario**, a partir de la cual podemos administrar bases de datos XML.
- **Un servicio que provee acceso a las bases de datos** que administra a otras aplicaciones externas. Como todos los servicios, puede iniciarse o detenerse.

Desde la **interfaz de usuario** de BaseX podrás realizar las siguientes operaciones:

- **Crear bases de datos:** cada base de datos BaseX lleva un identificador y está enlazada a un fichero externo XML con los datos que van a ser administrados en dicha base de datos.
- **Examinar los datos:** examinar los datos del documento XML con varias vistas distintas: en formato XML, en forma de árbol, en forma de mapa, etc.
- **Escribir consultar XQuery:** escribir consultas XQuery, ejecutarlas y obtener los resultados. XQuery es un lenguaje basado en SQL, pero adaptado a la consulta de bases de datos XML.

A través del **uso de BaseX como servicio**, otras aplicaciones pueden beneficiarse de las siguientes características:

- **Arquitectura Cliente/Servidor**, con soporte de transacciones seguras ACID, gestión de usuarios y autenticación.
- Acceso a **documentos XML de gran extensión**.
- Posibilidad de **envío de consultas XQuery** al servidor para obtener los datos deseados.
- **Alto rendimiento**.

A continuación, instalaremos BaseX y comenzaremos a trabajar con él.

Obtener e instalar BaseX

Puedes **obtener BaseX** de manera gratuita, ya que se trata de una aplicación que se distribuye bajo licencia de software libre permisiva.

Sigue los pasos del siguiente vídeo y tendrás instalado BaseX en tu equipo en muy poco tiempo.

<https://vimeo.com/telefonicaed/review/276179508/3f30068506>

Utilizar BaseX

Ya has instalado la aplicación BaseX en tu equipo, y es el momento de comenzar a utilizarla.

En este apartado, te ofrecemos un vídeo que podrás seguir para realizar las siguientes operaciones:

- Utilizar la interfaz de usuario de BaseX para crear una base de datos que enlace con el fichero *cruceros.xml*. El nombre de la base de datos será *crucerosBD*.
- Mostrar y ocultar los distintos paneles de la interfaz de usuario de BaseX.
- Examinar los datos de los cruceros con distintas vistas: mapa, árbol, xml, etc.
- Ejecutar consultas XQuery.
- Cerrar la base de datos.
- Abrir de nuevo la base de datos.

Ahora, te proponemos que visualices el vídeo y luego realices las mismas operaciones con tu recién instalada aplicación BaseX.

<https://vimeo.com/telefonicaed/review/276179515/f45cf58ba5>

Si has seguido el vídeo, verás que se han ejecutado varias consultas XQuery. No te preocupes si no comprendes el código de dichas consultas, ya que en la siguiente lección las estudiarás en profundidad. Tu único objetivo ahora es ver cómo BaseX te permite expresar dichas consultas y ejecutarlas.

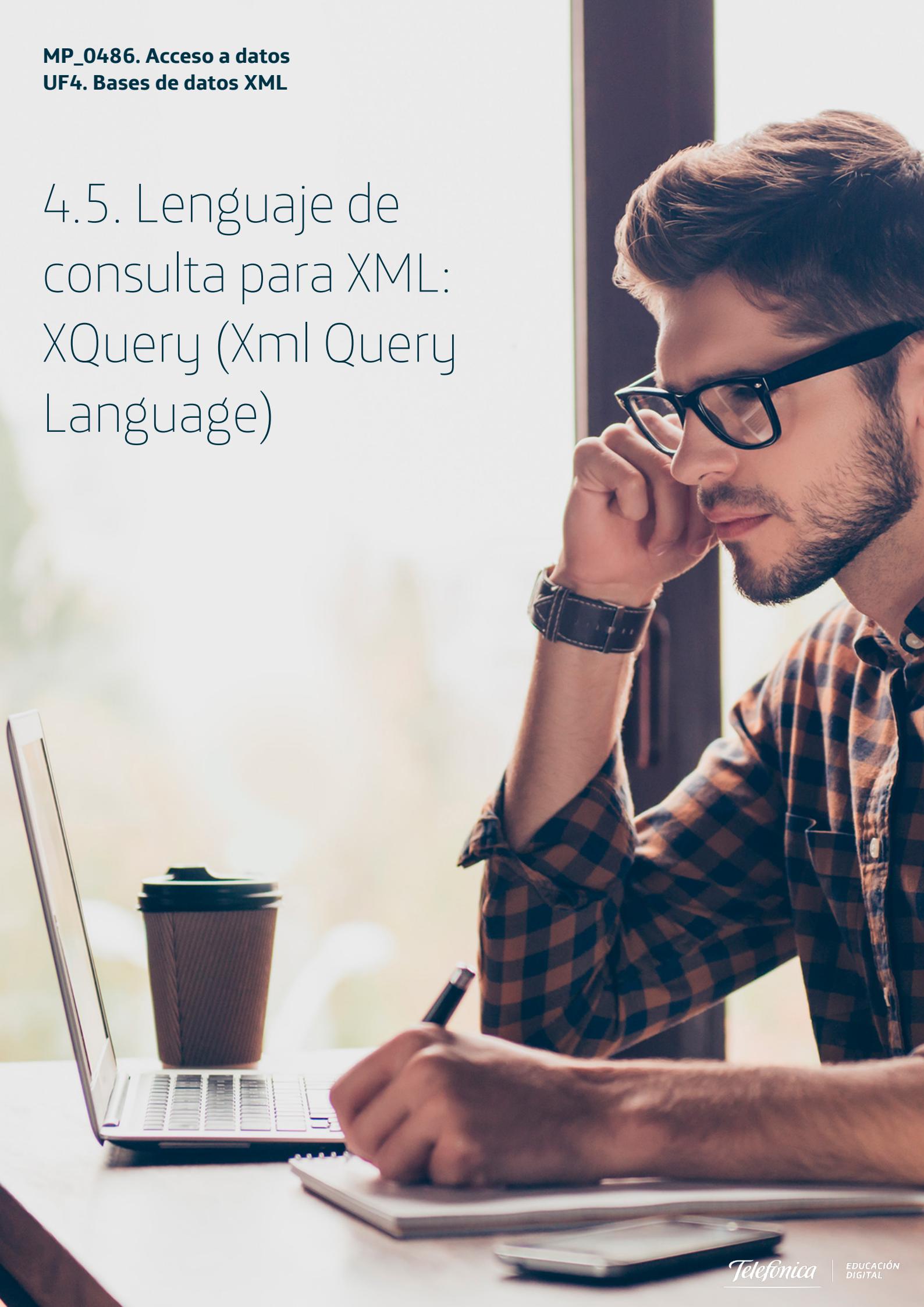
Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- **BaseX es un sistema de gestión de bases de datos XML** que ofrece dos herramientas principales: una interfaz de usuario, a partir de la cual podemos administrar bases de datos XML, y un servicio que provee a otras aplicaciones externas acceso a las bases de datos que administra. Como todos los servicios, puede iniciarse o detenerse.
- Desde la **interfaz de usuario** podemos crear bases de datos, examinarlas y ejecutar consultas XQuery.
- **BaseX como servicio** ofrece una arquitectura cliente/servidor que permite a otros programas acceso a las bases de datos que administra.

4.5. Lenguaje de consulta para XML: XQuery (Xml Query Language)



Índice

Objetivos	3
XQuery	4
Introducción a XQuery	4
Crear la base de datos ALMACEN.....	5
XPath	6
Where	7
Let	8
Order by	9
Despedida	10
Resumen.....	10

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Utilizar el lenguaje XQuery para formular consultas en bases de datos XML.
- Utilizar la aplicación BaseX para escribir y ejecutar las consultas XQuery.

XQuery

Introducción a XQuery

Desde enero de 2007, **XQuery** es el estándar para la consulta de bases de datos XML recomendado por W3C.

Estas son sus características principales:

- Su sintaxis se parece a SQL, pero incluye algunas características más propias del mundo de la programación.
- Se trata de un lenguaje que nos permite encontrar y extraer elementos dentro de un documento o base de datos XML.
- Cada consulta XQuery lee una secuencia de datos en formato XML y devuelve otra secuencia de datos XML con el resultado.
- XQuery, al igual que SQL, utiliza distintas cláusulas. Estas cláusulas siguen la norma FLWOR (de las iniciales *For*, *Let*, *Where*, *Order by* y *Return*).

La representación gráfica de una consulta XQuery quedaría así:



Representación de una consulta XQuery.

Vamos a analizar las distintas cláusulas de XQuery definidas por la palabra FLWOR.

Te aconsejamos que vayas probando los distintos ejemplos que aparecerán, abriendo la aplicación BaseX y la base de datos *cruceros*, tal y como aprendiste en la lección anterior.

- **FOR:** selecciona una secuencia de nodos por medio de una expresión XPath y el resultado lo almacena en una variable de memoria.

```
for $esc in /cruceros/crucero/escalas  
return $esc
```

\$esc es una variable de memoria cuyo contenido es el conjunto de nodos *escalas* obtenido a partir de la expresión XPath "/cruceros/crucero/escalas".

- **LET:** realiza un cálculo y asigna el resultado a una variable, que después puede ser utilizada en una condición *where*.

```
for $esc in /cruceros/crucero/escalas
let $c := count($esc//escala)
where $c < 7
return $esc
```

En el anterior ejemplo, para cada uno de los nodos *escalas*, cuenta el número de nodos hijo *escala* y va almacenando el resultado en la variable *\$c*, que posteriormente se utiliza en la condición *where*. El resultado final son los nodos *escalas* que tienen menos de 7 nodos hijo *escala*.

- **WHERE:** filtra los nodos.

```
for $esc in /cruceros/crucero/escalas/escala
where $esc/parada = "Venecia"
return $esc
```

Obtenemos los nodos *escala* con *parada* en Venecia.

- **ORDER BY:** ordena los nodos.

```
for $cru in /cruceros/crucero
order by $cru/destino
return $cru
```

Obtenemos los nodos *crucero* ordenados por *destino*.

- **RETURN:** en esta cláusula se define el resultado o salida de la consulta XQuery.

```
for $cru in /cruceros/crucero
return $cru/destino
```

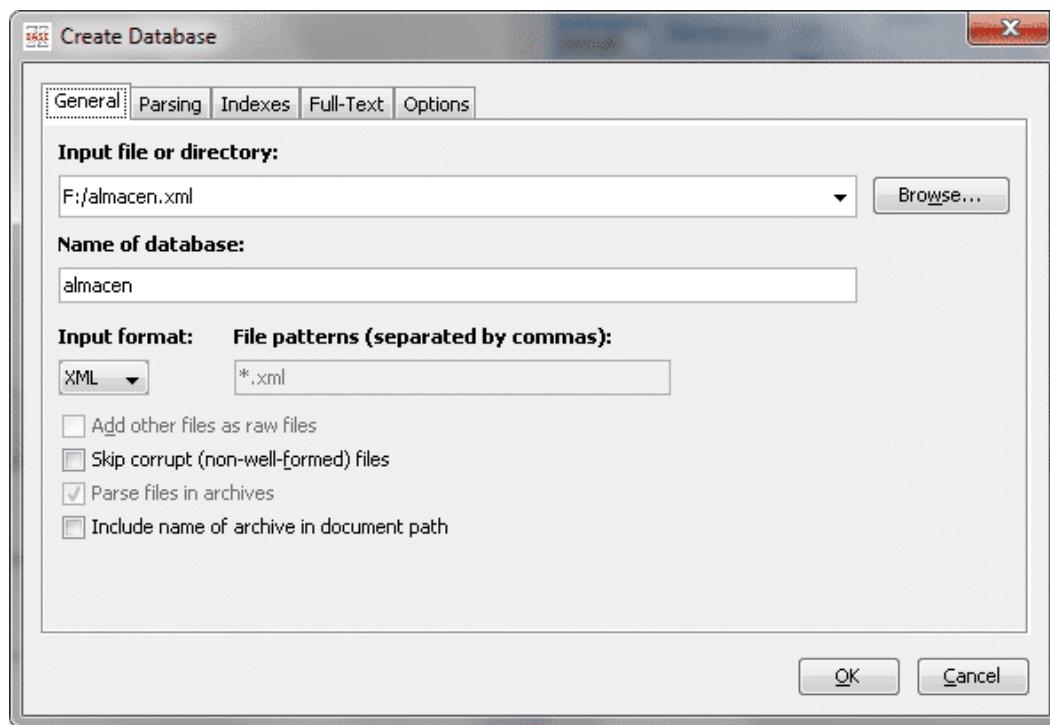
La variable *\$cru* contiene todos los nodos *crucero*, pero como salida de la consulta XQuery se devuelven sólo los nodos *destino*.

Crear la base de datos ALMACEN

A continuación vamos a trabajar con una **nueva base de datos, ALMACEN**, que representará un almacén de productos de alimentación.

Para realizar este ejemplo, puedes descargarte el archivo "*almacen.xml*" en la versión online de esta lección. Si ya tienes este archivo, sigue estos pasos para crear la nueva base de datos desde BaseX y familiarizarte con ella:

1. Abre la aplicación BaseX y selecciona *Database / New* en la barra de menús.



2. Utiliza el botón *Browse* para localizar el archivo *almacen.xml* en la ubicación que lo hayas guardado. Luego, escribe *almacen* en al campo *Name of database*.

Por último, pulsa el botón *OK*.

3. Dedica un rato a examinar los datos y familiarizarte con la estructura del nuevo documento. Comprobarás que se trata de un documento XML que contiene los datos de un almacén de productos de alimentación. Tiene un nodo raíz, denominado *almacen*, que consta de un conjunto de nodos *categoría*, cada uno de ellos compuesto por un conjunto de nodos *producto*.

You're the master of your life, the captain of your ship. Steer it with intention. Will you skirt the coast from one safe harbor to the next? Or will you sail into the vast open blue? Every day you get to decide anew what course to chart.

Ya conoces la estructura del documento XML. Manos a la obra y a escribir consultas XQuery.

XPath

XQuery utiliza XPath para seleccionar los nodos requeridos en la consulta.

Por eso, utiliza la aplicación BaseX para ejecutar cada uno de los ejemplos que te iremos mostrando.

1. Comencemos por un sencillo ejemplo que mostrará todos los nombre de las categorías:

```
for $cat in /almacen/categoría/nombreCategoria  
return $cat
```

El fragmento */almacen/categoría/nombreCategoria* es una expresión XPath. Hemos indicado toda la ruta dentro de la estructura del documento, hasta llegar al nodo *nombreCategoria*. Podemos simplificar la expresión del siguiente modo:

```
for $cat in //nombreCategoria  
return $cat
```

Recuerda que el símbolo *//* selecciona el nodo especificado, independientemente del nivel en que se encuentre dentro de la estructura XML.

2. Ahora, vamos a seleccionar todos los nodos *productos* que quedarán almacenados en la variable *\$pro*.

```
for $pro in /almacen/categoría/productos/producto  
return $pro
```

También podemos volver a utilizar XPath para seleccionar de nuevo elementos dentro del contenido de la variable *\$pro*.

```
for $pro in /almacen/categoría/productos/producto  
return $pro/nombreProducto
```

Where

Ahora vamos a completar nuestras consultas XQuery incluyendo la cláusula **WHERE**.

1. Comenzaremos por seleccionar aquellos productos cuyo precio sea superior a 30 euros.

```
for $pro in /almacen/categoría/productos/producto  
where $pro/precio > 30  
return $pro
```

Observa que la expresión condicional que va después de la cláusula *where* se expresa con la variable que contiene los datos previamente seleccionados, en este caso, *\$pro*. En la expresión condicional volvemos a utilizar XPath para acceder a los nodos deseados dentro de cada nodo *producto*.

2. Podemos utilizar los operadores lógicos *and*, *or* y *not* para expresar criterios más complejos. En el siguiente ejemplo vamos a obtener los productos cuyo precio esté en un rango entre 30 y 40 euros.

```
for $pro in /almacen/categoría/productos/producto
where $pro/precio >= 30 and $pro/precio <= 40
return $pro
```

3. El siguiente ejemplo devuelve los productos de la categoría *Bebidas*.

```
for $cat in /almacen/categoría
where $cat/nombreCategoria = "Bebidas"
return $cat/productos/producto
```

4. Ahora, vamos a obtener todos los artículos de las categorías *bebidas* o *condimentos*.

```
for $cat in /almacen/categoría
where $cat/nombreCategoria = "Bebidas" or $cat/nombreCategoria = "Condimentos"
return $cat/productos/producto
```

5. Y, por último, mostraremos todos los artículos que no sean ni *bebidas* ni *condimentos*.

```
for $cat in /almacen/categoría
where not($cat/nombreCategoria = "Bebidas" or $cat/nombreCategoria =
"Condimentos")
return $cat/productos/producto
```

Let

En ocasiones nos interesa filtrar los resultados según el valor de una expresión calculada.

Aquí es donde juega su papel la cláusula **Let**.

1. Si multiplicamos para cada producto el stock por el precio, nos dará como resultado una cantidad que corresponde a lo que podríamos recaudar por la venta de todo el stock de dicho producto.

Imagina que te piden obtener los productos cuya recaudación posible sea mayor a 2.000 euros; lo podrías resolver con la siguiente consulta que emplea la cláusula *Let*:

```
for $pro in /almacen/categoría/productos/producto
let $recaudacion := $pro/precio*$pro/stock
where $recaudacion > 2000
return $pro
```

2. También podemos utilizar **funciones de agregado**. El siguiente ejemplo muestra el nombre de todas las categorías que cuentan con más de 10 artículos distintos.

```
for $cat in /almacen/categoría
let $c := count($cat/productos/producto)
where $c > 10
return $cat/nombreCategoria
```

Order by

Ahora utilizaremos la cláusula **ORDER BY** para ordenar los resultados obtenidos por una consulta XQuery.

1. La siguiente consulta obtiene todos los **productos ordenados por nombre**.

```
for $pro in /almacen/categoría/productos/producto
order by $pro/nombreProducto
return $pro
```

2. En este otro ejemplo, ordenamos los artículos **en orden de menor a mayor importe** que podemos recaudar por sus ventas.

```
for $pro in /almacen/categoría/productos/producto
let $recaudacion := $pro/precio*$pro/stock
order by $recaudacion
return $pro
```

3. Y ahora, vamos a realizar la misma consulta, pero esta vez **en orden descendente**, es decir, de mayor a menor recaudación.

```
for $pro in /almacen/categoría/productos/producto
let $recaudacion := $pro/precio*$pro/stock
order by $recaudacion descending
return $pro
```

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Desde enero de 2007, **XQuery es el estándar para la consulta de bases de datos XML** recomendado por W3C.
- XQuery, al igual que SQL, **utiliza distintas cláusulas**. Estas cláusulas siguen la norma FLWOR (de las iniciales *For*, *Let*, *Where*, *Order by* y *Return*).

4.6. Trabajar con la librería de BaseX para Java



Índice

Objetivos	3
BaseX XQJ API.....	4
Introducción a BaseX XQJ API	4
Obtener BaseX XQJ API.....	5
Ejecutar consultas XQuery desde Java	6
Obtener objetos Node.....	11
Sentencias de actualización XQuery.....	12
Despedida	14
Resumen.....	14

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Desarrollar aplicaciones Java que accedan a bases de datos XML a través del API BaseX XQJ que la aplicación BaseX pone a nuestra disposición.
- Ejecutar consultas XQuery desde programas Java.

BaseX XQJ API

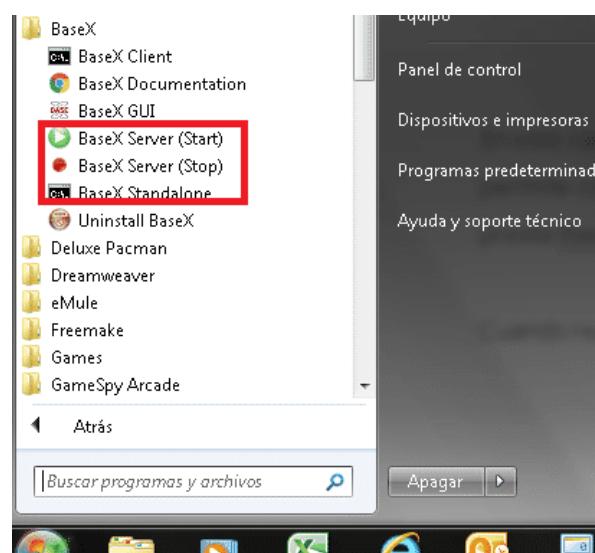
Introducción a BaseX XQJ API

Existen varias librerías o APIs en el mercado cuyo objeto es facilitar la explotación de documentos XML desde programas Java.

Su función principal es facilitar la **ejecución de consultas XQuery** y la obtención de los resultados para su posterior tratamiento.

En este caso, trabajaremos con la librería que nos brinda la aplicación BaseX (BaseX XQJ API) y que permite comunicar nuestro programa Java con el servicio de acceso a bases de datos XML, que presta BaseX en su modelo de arquitectura Cliente/Servidor.

Cuando realizaste la instalación de BaseX en tu equipo, además del ejecutable *BaseX GUI*, se instalaron otros programas, tal y como puedes ver en la imagen:



Otros programas instalados durante la instalación de BaseX.

Entre los programas instalados, hay que resaltar dos que tienen que ver especialmente con la arquitectura Cliente/Servidor de BaseX:

- **BaseX Server (Start):** inicia sesión con el servidor de BaseX, que quedará a la escucha de las peticiones de los clientes que requieren acceso a las bases de datos a través de un *host* y puerto.
- **BaseX Server (Stop):** cierra sesión con el servidor de BaseX.

Nuestra aplicación Java, haciendo uso de la librería BaseX XQJ API, actuará como cliente, realizando solicitudes al servidor de BaseX a través del *host* y el puerto.

Obtener BaseX XQJ API

Sigue los pasos que te vamos a exponer en este apartado para **obtener BaseX XQJ API**.

1. Accede a BaseX XQJ API

Pulsa el botón de la derecha para acceder a la web. <http://xqj.net/basex/>

2. Haz clic en el enlace *Download*.

The screenshot shows the homepage of the BaseX XQJ API. At the top, there's a navigation bar with links for Home, Products, Services, Community, Developer, and Contact. To the right of the navigation is a large red banner with the text "XQJ API". Below the banner, the page title "BaseX XQJ API 1.4" is displayed. To the right of the title is a sidebar with links for "The BaseX XQJ" (including Quick Start, Conformance Report, Compliance Definition Statement, and BaseX XQJ Examples), and "Download BaseX XQJ". The main content area contains a list of features and a "Quick Start" section. The "Quick Start" section includes four steps: 1. Download and install BaseX, 2. Launch BaseX via the "basexserver" service, 3. **Download the BaseX XQJ API.** (this step is highlighted with a red rectangle), and 4. Compile and run the following code.

3. Dentro de la lista de descargas, haz clic en la primera opción, que actualmente es la versión 1.4.0.

BaseX XQJ via ZIP Packages

If you're not using Maven, you can download a zip package which contains all nessacary jars to use BaseX XQJ. If you are planning on using BaseX XQJ in embedded mode, BaseX jars will also need to be included on the classpath at runtime.

Version	Build Date	Download	Size
1.4.0	11th February 2015	basex-xqj-1.4.0.zip	332 kb
1.3.0	10th February 2014	basex-xqj-1.3.0.zip	323 kb
1.2.3	1st October 2013	basex-xqj-1.2.3.zip	682 kb
1.2.2	4th September 2012	basex-xqj-1.2.2.zip	682 kb
1.2.1	23rd August 2012	basex-xqj-1.2.1.zip	682 kb
1.2.0	8th June 2012	basex-xqj-1.2.0.zip	677 kb

4. Espera a que se complete la descarga y guarda el archivo obtenido en la ubicación que deseas. Si no sabes dónde está el archivo, seguro que lo tendrás en la carpeta *descargas* de tu equipo. El archivo obtenido se llamará *basex-xqj-1.4.0.zip* o algo similar.

5. Descomprime el archivo para obtener una carpeta con nombre *basex-xqj-1.4.0*, donde estará ubicado el API.

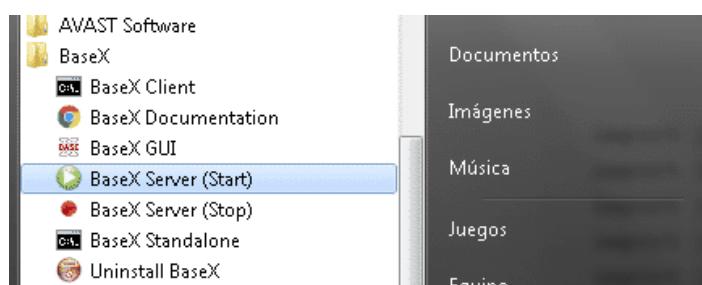
Ejecutar consultas XQuery desde Java

BaseX puede activarse como **un servicio para acceder de forma remota a las bases de datos XML** que administra.

Utilizaremos este servicio para obtener información sobre la base de datos ALMACEN, que creaste en la lección anterior.

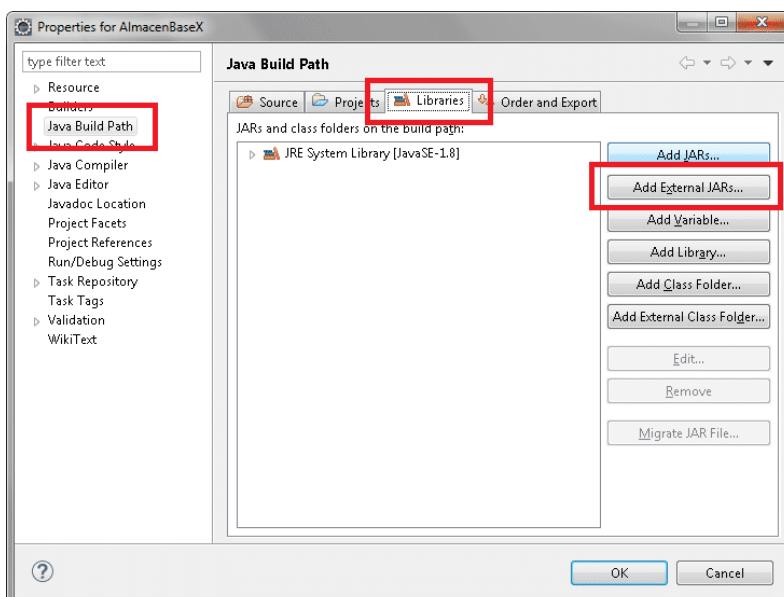
¡Ojo! Antes que nada, tendrás que dejar iniciado el servidor de BaseX que presta el servicio.

Accede a la carpeta de programas *BaseX* y ejecuta *BaseX Server (Start)*.

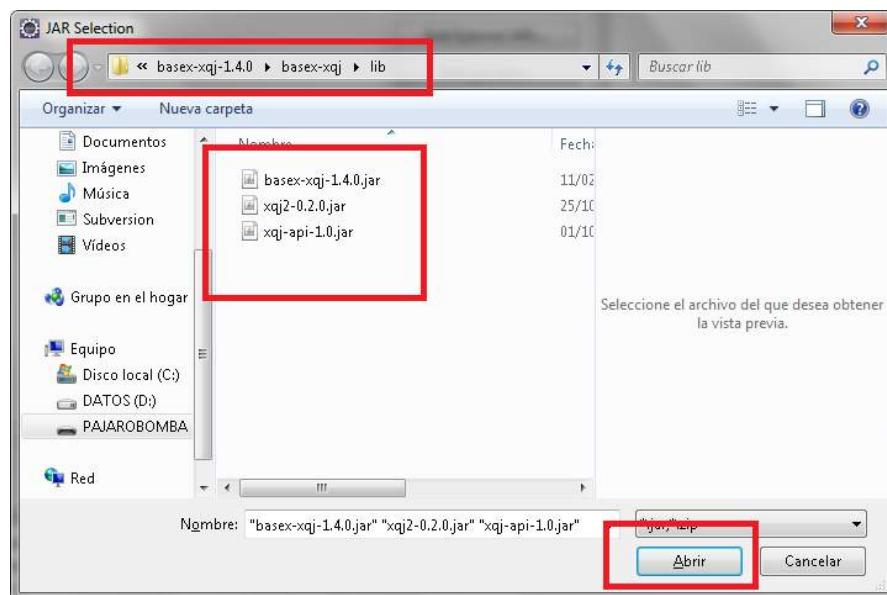


Una vez iniciado el servicio, sigue los pasos que te indicamos a continuación:

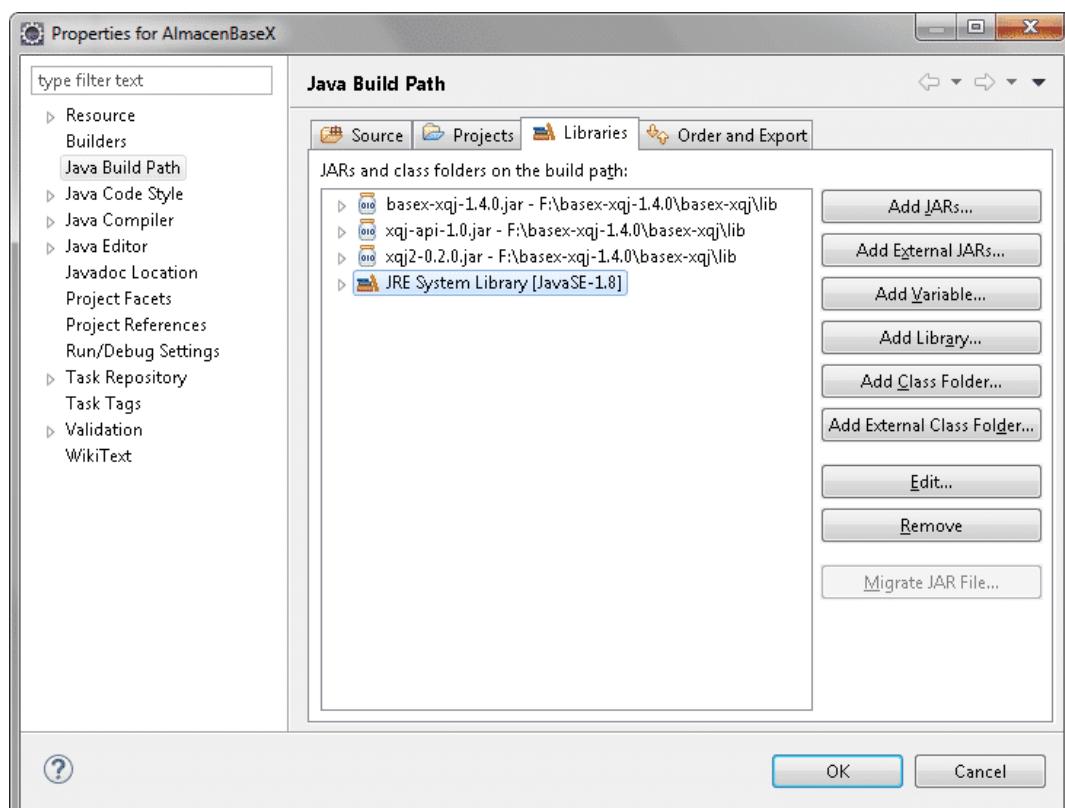
1. Crea un proyecto Java llamado *AlmacenBaseX* en Eclipse.
2. Dentro del proyecto, importa la librería *BaseX XQJ API* que descargaste en el apartado anterior. Para importar la librería, haz clic derecho sobre el nombre del proyecto y selecciona en el menú contextual la opción *Properties*. Dentro de las propiedades, tendrás que situarte en *Java Build Path* y en la pestaña *Libraries*.



Haz clic en el botón *Add External JARs...*. En el cuadro de diálogo *JAR Selection*, sitúate en la ubicación donde descargaste la librería y busca la carpeta *lib*. La ubicación será algo así: *baseX-xqj-1.4.0\baseX-xqj\lib*. Después, selecciona los tres archivos .jar que aparecerán y haz clic en el botón *Abrir*.



Finalmente, el cuadro de diálogo *Properties* quedará así:



3. Crea la clase *Principal* con el código que aparece a continuación; después, la analizaremos detenidamente para que comprendas cómo funciona la librería *BaseX XQJ API*.

```
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;
import javax.xml.xquery.XQResultSequence;
import net.xqj.baseX.BaseXXQDataSource;

public class Principal {
    public static void main(String[] args) {
        XQDataSource xds = new BaseXXQDataSource();
        XQConnection con;
        XQExpression expr;
        XQResultSequence result;
        String sentencia;

        try {
            xds.setProperty("serverName", "localhost");
            xds.setProperty("port", "1984");
            con = xds.getConnection("admin", "admin");
        } catch (XQException e) {
            System.out.println("Error al establecer la conexión con
BaseX");
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Establecida la conexión con BaseX");
        sentencia = "for $pro in fn:collection('almacen')//productos return
$pro/producto";

        try {
            expr = con.createExpression();
            result = expr.executeQuery(sentencia);
        } catch (XQException e) {
            System.out.println("Error al ejecutar la sentencia XQuery");
            System.out.println(e.getMessage());
            return;
        }

        try {
            while (result.next()) {
                System.out.println(result.getItemAsString(null));
            }
        } catch (XQException e) {
            System.out.println("Error al recorrer los elementos
obtenidos");
            System.out.println(e.getMessage());
        }

        try {
            con.close();
        } catch (XQException e) {
            System.out.println("Error al cerrar la conexión con BaseX");
            System.out.println(e.getMessage());
        }
    }
}
```

En primer lugar, observa que estamos utilizando nuevas clases, ubicadas en dos paquetes diferentes:

javax.xml.xquery

Procesador XQuery para Java que contiene los recursos necesarios para acceder a la base de datos XML requerida y ejecutar sentencias XQuery.

net.xqj.baseX

A este paquete pertenece la clase BaseXXQDataSource (origen de datos ligado a BaseX) que implementa la interfaz genérica XQDataSource.

Ahora, vamos a estudiar el código paso a paso:

Establecimiento de la conexión con BaseX:

```
XQDataSource xds = new BaseXXQDataSource();
XQConnection con;
XQExpression expr;
XQResultSequence result;
String sentencia;

try {
    xds.setProperty("serverName", "localhost");
    xds.setProperty("port", "1984");
    con = xds.getConnection("admin", "admin");
} catch (XQException e) {
    System.out.println("Error al establecer la conexión con BaseX");
    System.out.println(e.getMessage());
    return;
}
System.out.println("Establecida la conexión con BaseX");
```

Lo primero que hacemos es declarar una referencia a un objeto de tipo **XQDataSource**, pero **XQDataSource** no es una clase, sino una interfaz genérica que es implementada por varias clases, cada una de las cuales representa a un determinado gestor de bases de datos XML. **BaseXXQDataSource** es una clase directamente relacionada con BaseX; para otra aplicación se utilizaría otra clase diferente.

Con los métodos **setProperty()** estamos designando los valores para establecer la conexión con XBase. Estos valores son: **serverName**, que en nuestro caso es **localhost** y **port** o puerto por el que escucha BaseX, que es **1984**.

Por último, establecemos la conexión con el método **getConnection()** que recibe dos argumentos: **login** y **password** cuyos valores para BaseX son, por defecto, **admin** y **admin**.

```
sentencia = "for $pro in fn:collection('almacen')//productos return  
$pro/producto";  
  
try {  
    expr = con.createExpression();  
    result = expr.executeQuery(sentencia);  
} catch (XQException e) {  
    System.out.println("Error al ejecutar la sentencia XQuery");  
    System.out.println(e.getMessage());  
    return;  
}
```

La clase **XQExpression** nos permite ejecutar sentencias XQuery por medio de su método **executeQuery()**, que recibe como argumento una cadena con la sentencia a ejecutar y devuelve un objeto **XQResultSequence**, que nos permite acceso secuencial a los elementos resultado.

La expresión **XPath** de acceso a los elementos XML contiene la expresión **fn:collection('almacen')**. Con dicha expresión indicamos el nombre de la base de datos a la que deseamos acceder, puesto que sabemos que BaseX nos da acceso a distintas bases de datos.

```
try {  
    while (result.next()) {  
        System.out.println(result.getItemAsString(null));  
    }  
} catch (XQException e) {  
    System.out.println("Error al recorrer los elementos obtenidos");  
    System.out.println(e.getMessage());  
}
```

Nuestro objeto *result*, de tipo **XQResultSequence** dispone del método **next()** que nos permite ir avanzando al siguiente elemento o nodo dentro de la estructura XML resultado de la ejecución de la sentencia XQuery. Con el método **getItemAsString()** obtenemos todo el contenido XML de cada elemento; **getItemAsString()** recibe como argumento parámetros de serialización, pero en la mayoría de los casos es suficiente con pasar el valor *null*.

```
try {  
    con.close();  
} catch (XQException e) {  
    System.out.println("Error al cerrar la conexión con BaseX");  
    System.out.println(e.getMessage());  
}
```

Por último, debemos cerrar la conexión con BaseX.

Ahora puedes probar a modificar el contenido de la variable *sentencia* para poner a prueba otros ejemplos XQuery, tal y como aprendiste en la lección anterior.

Obtener objetos Node

Vamos a modificar el programa anterior para obtener, por cada producto iterado, un objeto *Node* que representará el árbol DOM de cada elemento.

Recuerda que la clase *Node* pertenece al *parser DOM* que estudiaste en la segunda lección de esta unidad.

```
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;
import javax.xml.xquery.XQResultSequence;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import net.xqj.basex.BaseXXQDataSource;
public class Principal {
    public static void main(String[] args) {
        XQDataSource xds = new BaseXXQDataSource();
        XQConnection con;
        XQExpression expr;
        XQResultSequence result;
        String sentencia;
        try {
            xds.setProperty("serverName", "localhost");
            xds.setProperty("port", "1984");
            con = xds.getConnection("admin", "admin");
        } catch (XQException e) {
            System.out.println("Error al establecer la conexión con BaseX");
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Establecida la conexión con BaseX");
        sentencia = "for $pro in fn:collection('almacen')//productos return
$pro/producto";
        try {
            expr = con.createExpression();
            result = expr.executeQuery(sentencia);
        } catch (XQException e) {
            System.out.println("Error al ejecutar la sentencia XQuery");
            System.out.println(e.getMessage());
            return;
        }
        int contador = 0;
        try {
            while (result.next()) {
                contador++;
                System.out.println("Producto nº " + contador);
                Node nodoProducto = result.getNode();
                mostrarProducto(nodoProducto);
            }
        } catch (XQException e) {
            System.out.println("Error al recorrer los elementos obtenidos");
            System.out.println(e.getMessage());
        }
        try {
            con.close();
        } catch (XQException e) {
            System.out.println("Error al cerrar la conexión con BaseX");
        }
    }
}
```

```

        System.out.println(e.getMessage());
    }

    private static void mostrarProducto(Node nodo) {
        NodeList nodos = nodo.getChildNodes();
        for (int i=0; i<nodos.getLength();i++) {
            Node nodoHijo = nodos.item(i);
            if (nodoHijo.getNodeType() == Node.ELEMENT_NODE) {
                System.out.println(nodoHijo.getNodeName() + ":" + 
nodoHijo.getTextContent());
            }
        }
    }
}

```

Sentencias de actualización XQuery

XQuery da soporte completo también para tareas de actualización de documentos XML, permitiendo no sólo la obtención de nodos, sino también su **borrado, actualización o inserción**.

Puedes realizar unas pruebas desde el editor XQuery de BaseX.

```

insert node
<categoria id="9">
  <nombreCategoria>Legumbres</nombreCategoria>
  <productos>
    <producto>
      <nombreProducto>Garbanzos del Bierzo</nombreProducto>
      <cantidadPorUnidad>Paquete de un kg</cantidadPorUnidad>
      <precio>20.16</precio>
      <stock>39</stock>
    </producto>
  </productos>
</categoria>
after /almacen/categoría[8]

```

Esta sentencia añade la nueva categoría *legumbres* con su producto "Garbanzos del Bierzo" y lo sitúa justo después de la octava categoría.

```

insert node
<producto>
  <nombreProducto>Judiones del Bierzo</nombreProducto>
  <cantidadPorUnidad>Paquete de un kg</cantidadPorUnidad>
  <precio>18.16</precio>
  <stock>12</stock>
</producto>
after /almacen/categoría[last()]/productos/producto[last()]

```

Esta sentencia añade el nuevo producto "Judiones del Bierzo" detrás del último producto de la última categoría que es *Legumbres*.

```
replace value of node  
/almacen/categoría[last()]//productos/producto[last()]/precio  
with 3
```

Esta sentencia asigna el valor 3 al precio del último producto de la última categoría.

```
delete node /almacen/categoría[last()]
```

Esta sentencia elimina la última categoría, que para nuestro documento es *Legumbres*.

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- **BaseX XQJ API** es una librería suministrada por BaseX que actúa como intermediaria entre un programa Java y las bases de datos que suministra el servidor de BaseX.
- La principal función de la librería BaseX XQJ API es permitir **ejecutar sentencias XQuery desde aplicaciones Java**.
- Podemos obtener la representación DOM de los elementos obtenidos a través de la ejecución de una sentencia XQuery como objeto de la clase *Node*.

5.1. Creación de componentes: propiedades y atributos



Índice

Objetivos	3
Clases, atributos y métodos	4
Crear clases personalizadas.....	4
Los atributos.....	5
Los métodos.....	7
Sobrecarga	8
Sobrecarga de constructores.....	9
Interfaz y modificaciones de acceso.....	9
Relaciones entre clases: jerarquía	11
Relaciones de agregación y composición	11
Agregación	11
Composición	12
Herencia.....	14
Ejemplo de herencia	15
Polimorfismo	17
Relación entre polimorfismo y excepciones.....	18
Uso de objetos de excepción.....	18
Tipos de excepciones (clases derivadas de <i>Exception</i>).....	19
¿Cómo se trabaja con las excepciones comprobadas?	22
Provocando excepciones no comprobadas.....	23
Varios catch	25
<i>Multicatch</i>	27
Polimorfismo y excepciones	28
Relanzamiento o propagación de excepciones	29
Resumen de las excepciones más habituales	32
Despedida	34
Resumen.....	34

Objetivos

Los temas tratados en esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.

Con esta lección perseguimos los siguientes objetivos:

- Crear componentes software formados por jerarquías de clases Java construidas utilizando todas las características de la programación orientada a objetos: agregación, composición, herencia, abstracción y polimorfismo.
- Relacionar el concepto de polimorfismo con la gestión de excepciones.

Clases, atributos y métodos

Crear clases personalizadas

Las clases son abstracciones de entidades del mundo real que sirven como modelos para construir objetos basados en ellas.

Este es el formato de construcción de una clase:

```
modificadores class nombreClase {  
    // Cuerpo de la clase  
}
```

Donde los modificadores definen el tipo de visibilidad y comportamiento de la clase.

Los modificadores que pueden aplicarse a una clase son:

- **final:** la clase no puede ser heredada por otras clases.
- **public:** la clase es visible dentro de cualquier paquete del proyecto.
- **protected:** la clase es visible solo dentro del mismo paquete donde está definida.
- **private:** la clase solo es visible por otra clase que la contenga.
- **abstract:** la clase es abstracta. No se pueden construir objetos a partir de ella. Está pensada solo para ser heredada.

Posiblemente te estés preguntando por qué existe un modificador de tipo **private** asociado a una clase. **¿Quién quiere una clase que no es accesible?**

Pues la respuesta está en las clases internas. A continuación te muestro un ejemplo:

```
public class Casa {  
    private int metros;  
    private int numHabitaciones;  
    private Propietario propietario;  
  
    public Casa(int metros, int numHabitaciones, String nombre, String  
telefono) {  
        this.metros = metros;  
        this.numHabitaciones = numHabitaciones;  
        this.propietario = new Propietario(nombre, telefono);  
    }  
  
    @Override  
    public String toString() {  
        return "Casa [metros=" + metros + ", numHabitaciones=" +  
            numHabitaciones + ", propietario=" +  
            propietario.toString() + "]";  
    }  
}
```

```
private class Propietario {  
    private String nombre;  
    private String telefono;  
  
    public Propietario(String nombre, String telefono) {  
        super();  
        this.nombre = nombre;  
        this.telefono = telefono;  
    }  
  
    @Override  
    public String toString() {  
        return "Propietario [nombre=" + nombre + ", telefono=" +  
telefono  
                + "]";  
    }  
}
```

Observa que dentro de la clase *Casa* tenemos una clase interna llamada *Propietario*. Esta clase es privada, lo que significa que solo es visible por su clase contenedora *Casa*.

Puedes ponerlo en práctica con la siguiente clase *Principal*.

```
public class Principal {  
    public static void main(String[] args) {  
        Casa unaCasa = new Casa(100, 3, "Pepe", "Pérez");  
        System.out.println(unaCasa);  
    }  
}
```

Los atributos

Las clases tienen atributos. Los atributos pueden pertenecer al objeto (propiedades del objeto) o a la clase (propiedades que pertenecen a la clase).

Formato de creación de un atributo:

```
modificadores tipo nombreAtributo [= valor];
```

Los atributos admiten los siguientes modificadores:

private

El atributo es accesible solo dentro de la clase donde está declarado.

public

El atributo es accesible desde cualquier clase situada en cualquier paquete y también por las clases derivadas.

protected

El atributo es accesible solo desde otras clases situadas en el mismo paquete y también por las clases derivadas, independientemente del paquete en que se encuentren.

default

Equivale a no poner ningún modificador de acceso. El atributo es accesible solo desde otras clases situadas en el mismo paquete y desde clases derivadas, siempre que se encuentren en el mismo paquete.

static

El atributo pertenece a la clase, no al objeto. El valor del atributo es compartido por todos los objetos creados de la clase.

final

El atributo es una constante. Su valor no puede ser modificado.

Veamos un ejemplo:

```
public class Triangulo {  
    // Constante publica, su valor no puede ser modificado.  
    // Además pertenece a la clase, no al objeto  
    public static final float PI = 3.141592F;  
  
    // Propiedades privadas.  
    private int lado1;  
    private int lado2;  
    private int lado3;  
}
```

Los atributos *lado1*, *lado2* y *lado3* pertenecen al objeto. Si creamos tres objetos de la clase *Triangulo*, cada uno de ellos tendrá distintos lados, es decir, existirán tres grupos de estas propiedades.

La variable *PI*, sin embargo, pertenece a la clase. Aunque creamos 20 objetos de la clase *Triangulo*, habrá un solo atributo *PI* compartido para todos ellos. Además *PI* es una constante, su valor no puede ser modificado.

Las propiedades *lado1*, *lado2* y *lado3*, además, son privadas, lo que significa que para tener acceso a ellas a través de los objetos, necesitamos implementar los métodos públicos *getLado1()*, *getLado2()* y *getLado3()*.

Los métodos

Las clases sirven como abstracciones de objetos que además de tener atributos (características) pueden realizar acciones.

Las acciones se implementan por medio de los métodos.

Los métodos tienen la siguiente estructura:

```
modificadores tipo nombreMetodo([parámetros]) {  
}
```

- Los **modificadores** definen el tipo de visibilidad y comportamiento del método.
- El **tipo** define el tipo de valor que retorna el método. Se pone *void* si el método no retorna ningún valor.
- Los **parámetros** definen los datos que debe recibir el método.

Los modificadores aplicables a un método son:

private

El método es accesible solo dentro de la clase donde está declarado.

public

El método es accesible desde cualquier clase situada en cualquier paquete y también por las clases derivadas.

protected

El método es accesible solo desde otras clases situadas en el mismo paquete y también por las clases derivadas, independientemente del paquete en que se encuentren.

default

Equivale a no poner ningún modificador de acceso. El método es accesible solo desde otras clases situadas en el mismo paquete y desde clases derivadas siempre que se encuentren en el mismo paquete.

static

El método pertenece a la clase, no al objeto. Los métodos estáticos solo pueden acceder a atributos estáticos, nunca a propiedades del objeto.

final

El método no puede ser sobrescrito por las clases derivadas.

Ejemplo:

```
public final String verTipoTriangulo() {
    if (this.lado1 == this.lado2 && this.lado2 == this.lado3) {
        return "Equilátero";
    } else if (this.lado1 == this.lado2 || this.lado2 == this.lado3
               || this.lado1 == this.lado3) {
        return "Isósceles";
    } else {
        return "Escaleno";
    }
}
```

Analicemos un poco el método:

- El método ***verTipoTriangulo()*** pertenece al objeto, es decir, trabaja con las propiedades de un triángulo concreto.
- El método ***verTipoTriangulo()*** no podrá ser sobrescrito por otras clases que hereden de *Triangulo*, ya que está declarado como *final*.

Sobrecarga

La sobrecarga de métodos consiste en definir varias implementaciones para el mismo método, de modo que pueda ser utilizado de múltiples maneras. Veamos un ejemplo:

```
public class Compra {
    private String descripcion;
    private float precio;
    private float cantidad;

    public Compra(String descripcion, float precio, float cantidad) {
        this.descripcion = descripcion;
        this.precio = precio;
        this.cantidad = cantidad;
    }

    public float calcularPrecioVenta() {
        return this.cantidad * this.precio;
    }

    public float calcularPrecioVenta(float descuento) {
        return this.cantidad * this.precio - this.cantidad * this.precio
               * descuento;
    }

    @Override
    public String toString() {
        return "Compra [descripcion=" + descripcion + ", precio=" + precio
               + ", cantidad=" + cantidad + "]";
    }
}
```

La clase *Compra* define dos implementaciones distintas para el método *calcularPrecioVenta()*, uno sin parámetros y otro que recibe un porcentaje de descuento.

Cada una de las implementaciones debe variar en el número y/o tipo de parámetros.

Ahora podemos calcular el precio de una compra de dos maneras distintas.

```
public class Principal {  
    public static void main(String[] args) {  
        Compra miCompra = new Compra("Peras", 1.25F, 3.5F);  
        System.out.println(miCompra.toString());  
        System.out.println("Precio: " + miCompra.calcularPrecioVenta());  
        System.out.println("Precio con descuento: "  
                           + miCompra.calcularPrecioVenta(0.1F));  
    }  
}
```

Sobrecarga de constructores

Recuerda que el método constructor es un método especial que lleva el mismo nombre que la clase y es invocado en el momento de construir el objeto. También puede estar sobrecargado, así tenemos más posibilidades a la hora de construir los objetos.

```
public class Compra {  
    private String descripcion;  
    private float precio;  
    private float cantidad;  
  
    public Compra(String descripcion, float precio, float cantidad) {  
        this.descripcion = descripcion;  
        this.precio = precio;  
        this.cantidad = cantidad;  
    }  
  
    public Compra(String descripcion, float precio) {  
        this.descripcion = descripcion;  
        this.precio = precio;  
        this.cantidad = 1;  
    }  
  
    .....  
}
```

Ahora podemos construir un objeto de la clase *Compra* de dos maneras, indicando la cantidad o sin indicar la cantidad, en cuyo caso asigna una unidad por defecto.

Interfaz y modificaciones de acceso

La interfaz de la clase está definida por lo que dejamos ver al exterior, de modo que está determinada por los modificadores de visibilidad que utilizamos.



La interfaz de un televisor son los botones que tenemos disponibles para interactuar con ella.

De la misma forma, la interfaz de los objetos de la clase *Compra* son las propiedades y los métodos públicos a los que el usuario puede acceder a través del objeto.

Relaciones entre clases: jerarquía

Relaciones de agregación y composición

La agregación y la composición son tipos de relaciones entre clases donde una propiedad de una clase es un objeto de otra clase.

Por ejemplo: un objeto *coche* podría tener las propiedades *matricula*, *marca*, *modelo* y *motor*, siendo *motor* un objeto con sus propiedades *tipoCombustible*, *caballos*, etc.

Entonces, ¿dónde está la diferencia entre agregación y composición?

La diferencia entre agregación y composición es bastante sutil.

Agregación

Cuando existe una relación de agregación entre dos clases, **al eliminar la clase contenedora, no se elimina la clase contenida**.

Verás un ejemplo en el que una clase *Examen* agrega un objeto de la clase *Alumno*.

Las relaciones de agregación también se denominan de “**composición débil**”.

```
public class Alumno {  
    private String nombre;  
    private String telefono;  
  
    public Alumno(String nombre, String telefono) {  
        this.nombre = nombre;  
        this.telefono = telefono;  
    }  
  
    @Override  
    public String toString() {  
        return "Alumno: " + this.nombre + " - " + this.telefono;  
    }  
}
```

```
public class Examen {  
    private Alumno alumno;  
    private String asignatura;  
    private float nota;  
  
    public Examen(Alumno alumno, String asignatura, float nota) {  
        this.alumno = alumno;  
        this.asignatura = asignatura;  
        this.nota = nota;  
    }  
}
```

```
public Alumno getAlumno() {  
    return alumno;  
}  
  
@Override  
public String toString() {  
    return this.alumno.toString() + "" + this.asignatura + ":" +  
this.nota;  
}  
}
```

Observa que el constructor de *Examen* recibe un objeto de tipo *Alumno* que podrá ser creado externamente y pasado como argumento.

Ahora puedes ponerlo en práctica con la clase *Principal*:

```
public class Principal {  
    public static void main(String[] args) {  
        Alumno al = new Alumno("Miguel Pérez", "616669966");  
        Examen ex = new Examen(al, "Matemáticas", 7.5F);  
        System.out.println(ex.toString());  
        // Acceso al alumno a partir del examen.  
        System.out.println(ex.getAlumno().toString());  
        // Acceso al alumno como objeto autónomo  
        System.out.println(al.toString());  
    }  
}
```

Observa que hemos creado dos objetos (*al* y *ex*) de las *clasesAlumno* y *Examen*. El objeto *al* es completamente autónomo con respecto al objeto *ex*. No depende de él para su subsistencia.

El objeto *al* podría servir de componente para otros objetos diferentes que lo requieran.

Composición

Cuando existe una relación de composición entre dos clases, **al eliminar la clase contenedora, se elimina la clase contenida.**

Para el ejemplo expuesto anteriormente, eliminar el objeto *Examen* supone la eliminación de su objeto *Alumno* contenido.

Las relaciones de composición también se denominan de "**composición fuerte**".

Si quieras ponerlo en práctica puedes realizar una copia del proyecto anterior con otro nombre y realizar los cambios necesarios.

La clase *Alumno* no cambia.

Ahora modifica la clase *Examen* de la siguiente forma:

```
public class Examen {  
    private Alumno alumno;  
    private String asignatura;  
    private float nota;  
  
    public Examen(String asignatura, float nota, String nombre, String tlf) {  
        this.alumno = new Alumno(nombre, tlf);  
        this.asignatura = asignatura;  
        this.nota = nota;  
    }  
  
    public Alumno getAlumno() {  
        return alumno;  
    }  
  
    @Override  
    public String toString() {  
        return this.alumno.toString() + " " + this.asignatura + ":" +  
this.nota;  
    }  
}
```

Observa que ahora, en lugar de pasar como argumento al constructor un objeto *Alumno* ya creado, le pasamos los valores de nombre y teléfono para crear el objeto *Alumno* dentro del código interno de la clase *Examen*.

Para cada objeto *Examen* construido, se creará un objeto *Alumno* internamente.

El *Alumno*, al haberse construido dentro del ámbito del *Examen*, compartirá con él dicho ámbito y vigencia (tiempo de vida).

Como has podido comprobar, la diferencia entre agregación y composición es muy sutil y depende de la técnica de programación utilizada.

Ahora, la clase *Principal* quedará así:

```
public class Principal {  
    public static void main(String[] args) {  
        Examen ex = new Examen  
            ("Matemáticas", 7.5F, "Miguel Pérez", "616669966");  
        System.out.println(ex.toString());  
        System.out.println(ex.getAlumno().toString());  
    }  
}
```

Herencia

La herencia es una característica de la programación orientada a objetos que permite crear una clase que hereda las propiedades y métodos de otra clase madre.

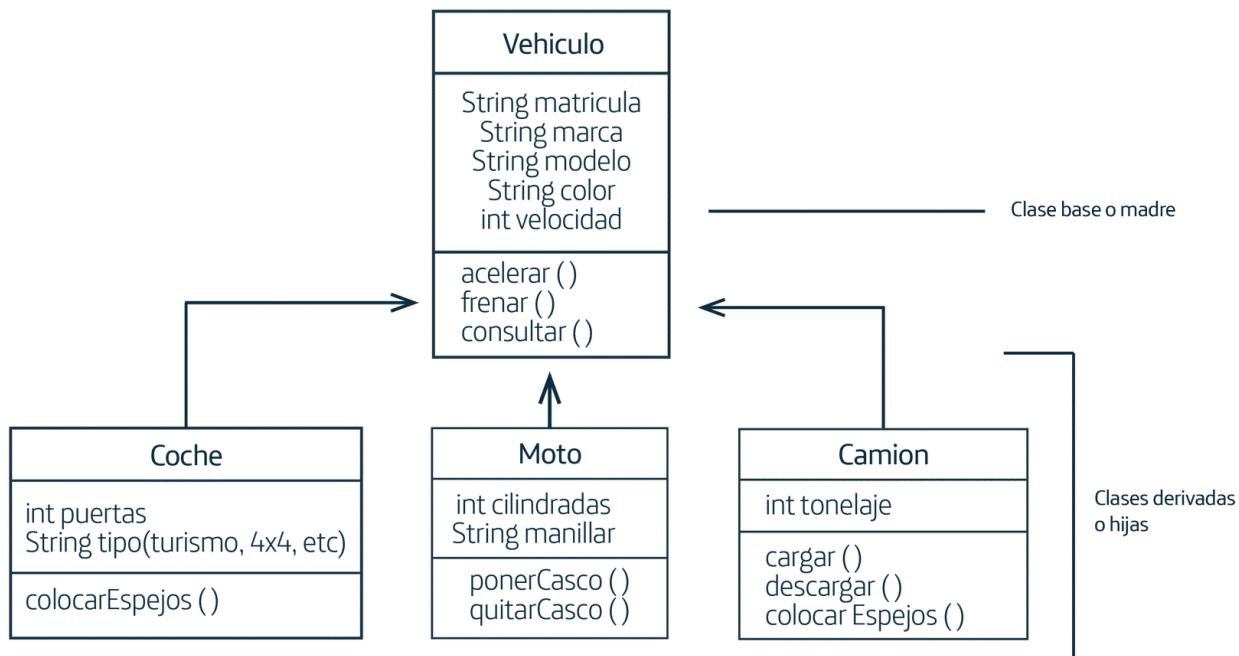
En la herencia se distinguen dos tipos de clase:

- Clase **base** o clase madre.
- Clase **derivada** o clase hija, que hereda las características de la madre.

La herencia es uno de los aspectos más importantes del paradigma de programación orientada a objetos y la distingue de otros paradigmas de programación.

Una clase hija es una especialización de la clase madre, por ejemplo: un coche es una especialización de un vehículo, pasamos de algo más abstracto a algo más concreto.

Tiene sentido aplicar la herencia a la hora de diseñar una nueva clase cuando existe una relación de tipo “**es un**”.



- Un coche **es un** vehículo.
- Una moto **es un** vehículo.
- Un camión **es un** vehículo.

Ejemplo de herencia

Vamos a crear una clase llamada *Persona* con la misma estructura de la anterior clase *Alumno*.

```
public class Persona {  
    private String nombre;  
    private String telefono;  
  
    public Persona(String nombre, String telefono) {  
        this.nombre = nombre;  
        this.telefono = telefono;  
    }  
  
    @Override  
    public String toString() {  
        return this.nombre + " - " + this.telefono;  
    }  
}
```

Ahora vamos a construir las clases derivadas *Alumno* y *Profesor*.

```
public class Alumno extends Persona {  
    private int numMatricula;  
  
    public Alumno(String nombre, String telefono, int numMatricula) {  
        super(nombre, telefono);  
        this.numMatricula = numMatricula;  
    }  
  
    @Override  
    public String toString() {  
        return "Alumno: " + super.toString() +  
            " Nº matrícula " + this.numMatricula + "]";  
    }  
}
```

```
public class Profesor extends Persona {  
    private String especialidad;  
  
    public Profesor(String nombre, String telefono, String especialidad) {  
        super(nombre, telefono);  
        this.especialidad = especialidad;  
    }  
  
    @Override  
    public String toString() {  
        return "Profesor: " + super.toString() + " Especialidad "  
            + this.especialidad;  
    }  
}
```

Vamos a analizar el código:

1. El constructor de una clase derivada se complica un poquito. El método constructor de *Alumno* no solo tiene que ocuparse de inicializar los valores de sus propiedades (en este caso la propiedad *numMatricula*), sino que también debe suministrar al constructor de la clase madre *Persona* los valores que necesita. Para ello se invoca al constructor de la clase madre de la siguiente manera:

super(argumentos);

Para nuestro ejemplo:

super(nombre, telefono);

El constructor de *Alumno* tiene 3 parámetros, los 2 primeros son suministrados al constructor de la clase *Persona* y el último es asignado directamente al constructor de la clase *Alumno*.

La palabra *super* debe ser lo primero que aparezca en el constructor.

2. La clase *Alumno* también ha heredado el método *toString()* de *Persona* y lo ha sobrescrito para especializarlo más. A su vez, en la clase *Persona* ya estaba sobrescrito del original de la clase *Object* (superclase de la que heredan todas las clases).

```
@Override
public String toString() {
    return "Alumno " + super.toString() + " Nº matrícula = " + this.numMatricula + "]";
}
```

La anotación **@Override** informa de que este método ya existe en la clase base (*Persona*) pero está siendo sobrescrito en la clase derivada. **Entonces, ¿cuál de los dos métodos ejecuta cuando lo invocamos desde la clase *Alumno*?** La respuesta está de nuevo en la utilización de la palabra *super*, que nos permite el acceso a los métodos de la clase base.

super.toString() - Ejecuta el método *toString()* de *Persona*.

toString() - Ejecuta el método *toString()* de *Alumno*.

Ahora, en la clase *Principal* puedes crear objetos más especializados para representar un alumno o un profesor en lugar de una persona sin más.

```
public class Principal {
    public static void main(String[] args) {
        Profesor pro = new Profesor("Luis Pérez", "913332211",
"Matemáticas");
        Alumno alu = new Alumno("Alicia Robles", "914445566", 3899);
        System.out.println(pro.toString());
        System.out.println(alu.toString());
    }
}
```

Polimorfismo

En el contexto de la programación orientada a objetos, el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos y se puede lograr de varias formas.

Permite que una referencia a un objeto pueda apuntar a distintos tipos de objetos. Una referencia a un objeto de tipo *Figura* puede apuntar a objetos de tipo *Triangulo*, *Rectangulo* o *Circulo*, igual que una referencia a un objeto *Animal* puede apuntar a un objeto *Pulga* o *Tiranosauro*.

Para nuestro ejemplo anterior, podemos tener una referencia a un objeto *Persona* que apunte indistintamente a un objeto *Profesor* o a un objeto *Alumno*.

El polimorfismo facilita la reutilización de código, ya que podemos utilizar el mismo método para procesar distintos tipos de objetos.

```
public class Principal {  
    public static void main (String[] args) {  
        Profesor pro = new Profesor("Luis Pérez", "913332211", "Matemáticas");  
        Alumno alu = new Alumno("Alicia Robles", "914445566", 3899);  
        procesar(pro);  
        procesar(alu);  
    }  
  
    public static void procesar(Persona p) {  
        System.out.println(p.toString());  
        System.out.println(p.saludar());  
    }  
}
```

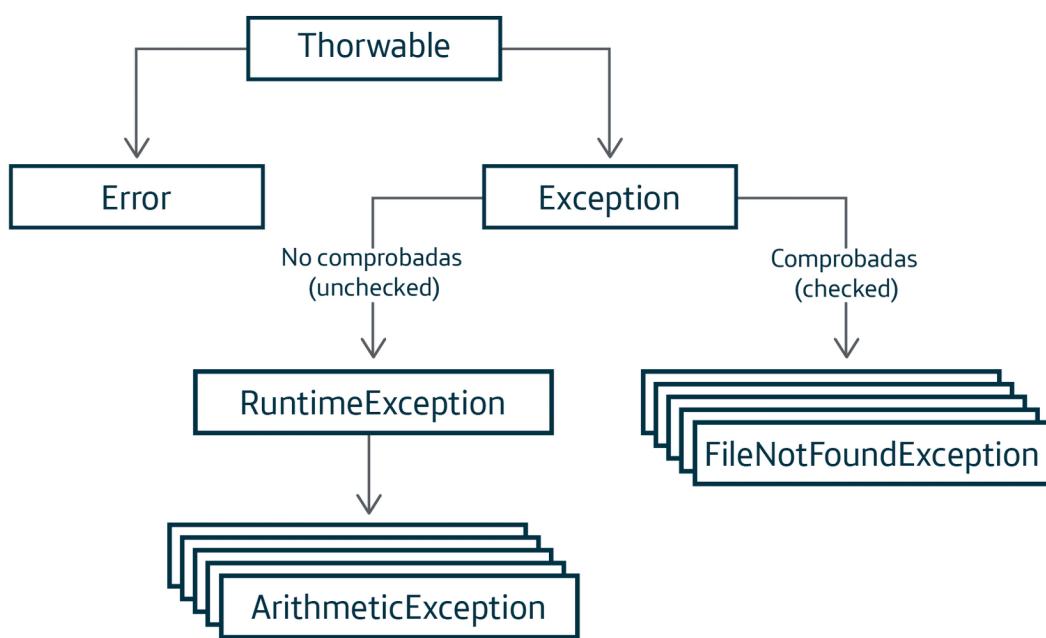
El método *procesar* es capaz de trabajar igualmente con un objeto *Profesor* que con un objeto *Alumno*, ya que ambos son de tipo *Persona*.

Esta es una de las grandes ventajas del polimorfismo.

Relación entre polimorfismo y excepciones

Uso de objetos de excepción

Cuando se produce una condición de error (excepción para Java), la máquina virtual genera un tipo de objeto especial con información sobre el suceso ocurrido; un objeto de tipo *Exception*. La jerarquía de clases para la gestión de excepciones en Java se puede resumir con la siguiente imagen:



Jerarquía de clases de excepción en Java.

La clase principal de la que heredan todas las clases que representan excepciones es **Throwable**.

Observando la estructura jerárquica de la imagen, puedes comprobar que *Throwable* se descompone en dos ramas, que representan dos situaciones de error:

- **Error:** representan situaciones que se escapan al control del programador, por lo que no deberíamos hacer nada con ellas. Por ejemplo, un error grave en el funcionamiento de la máquina virtual de Java que escapa a nuestro control.
- **Exception:** representan situaciones que el programador sí puede gestionar y por lo tanto, será de este tipo de clases de las que nos ocupemos en esta lección. Los objetos de este tipo son los que realmente denominamos excepciones.

Ejemplo de programa que genera un error o excepción

Prueba a ejecutar este pequeño programa, donde realizamos una operación claramente incorrecta; una división cuyo divisor es un cero:

```
public class Principal {  
    public static void main(String args[]) {  
        int cero=0;  
        int resul=6/cero;  
        System.out.println(resul);  
    }  
}
```

Programa que genera una situación de error o excepción.

Como resultado has obtenido un mensaje de error o excepción como el siguiente:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero  
at DivCero.main(DivCero.java:4)
```

La máquina virtual de Java (JVM) ha generado un objeto de la clase *ArithmetricException* y ha abortado el programa. A lo largo de la unidad aprenderás cómo puedes capturar dicho objeto para recuperar el control sobre la ejecución del programa y evitar que aborte la ejecución.

Tipos de excepciones (clases derivadas de *Exception*)

Volviendo de nuevo a observar la imagen, puedes comprobar que existen dos tipos de clases que derivan de *Exception*:

- **Excepciones no comprobadas que derivan de *RuntimeException*:** no estamos obligados a controlar estas excepciones. Tienen que ver con situaciones en que existe un mal planteamiento en el código. Puede evitarse que lleguen a ocurrir y no hay necesidad de tener que controlarlas, aunque podemos hacerlo. Nuestro ejemplo anterior de la división entre cero ha provocado una exception de tipo *ArithmetricException*, este es un ejemplo de excepción que no es obligatorio controlar. También está claro que podría evitarse esta situación antes de realizar la división.
- **Excepciones comprobadas que derivan directamente de *Exception*:** son situaciones de error de las que podemos recuperarnos, pero que no son fruto de un mal planteamiento del código, sino de una situación inesperada, tal como cuando abrimos un fichero de texto y de repente alguien ha borrado dicho fichero del disco. Debemos controlar este tipo de excepciones para que el programa pueda recuperarse de la situación de error sin necesidad de abortarlo.

¿Y cómo se controlan las excepciones?

Un bloque de código susceptible de producir una excepción debe encerrarse en un bloque **try** {...} seguido de un bloque **catch** {...}, que capturará el objeto de excepción. Debes utilizar el siguiente formato:

```
try {  
    // Sentencias que pueden provocar excepción  
}  
catch(Exception e){  
    // Respuesta a la situación de excepción  
}  
finally {  
    // Sentencias que se ejecutan incondicionalmente  
}
```

Formato *try ... catch ... finally*.

- **Bloque try:** donde se encierran las instrucciones que pueden provocar una situación de excepción, ya sea comprobada o no comprobada. La diferencia está en que si se trata de excepciones comprobadas, es obligatorio usar un bloque *try* y si se trata de excepciones no comprobadas, el uso de *try* es opcional.
- **Bloque catch:** donde se captura el objeto de excepción. Si dentro del bloque *try* ocurre una situación de excepción, el control pasa al siguiente bloque *catch*, capaz de recoger dicha excepción.
- **Bloque finally:** este bloque es opcional y contiene sentencias que se ejecutarán de manera incondicional ocurra o no una excepción.

```
public class Principal {  
    public static void main(String args[]) {  
        int cero=0;  
        int resul;  
        try {  
            resul=6/cero;  
            System.out.println(resul);  
        }  
        catch (ArithmaticException e) {  
            System.out.println("Se ha producido una excepción");  
            System.out.println(e.getMessage());  
        }  
        finally {  
            System.out.println("Hasta pronto");  
        }  
    }  
}
```

Paso a paso

Vamos ver paso a paso lo que ocurre con el programa anterior:

- La sentencia `resul=6/cero;` provoca una excepción, por lo que la máquina virtual de Java genera un objeto de tipo `ArithmetricException`. Al encontrarse dentro de un bloque `try`, el control pasará al primer bloque `catch` que pueda recoger un objeto de tipo `ArithmetricException` (pueden existir varios bloques `catch`). En el ejemplo, el objeto es recogido en la variable `e`.
- Dentro del bloque `catch`, el programa informa de lo ocurrido y utiliza el método `getMessage()` de la clase `Exception` para mostrar la descripción del error o excepción. Recuerda que `ArithmetricException` hereda de `Exception`, luego el objeto `e` es tanto una `ArithmetricException` como una `Exception`.
- La sentencia `System.out.println(resul);` será pasada por alto, no llegará a ejecutarse, ya que tras ocurrir la excepción en la línea anterior, el control pasó al bloque `catch`.
- Por último se ejecutará el bloque `finally`.

¿Y para qué sirve el bloque `finally`? Vas a descubrirlo con una pequeña práctica.

Pega este pequeño programa en un proyecto Eclipse y ejecútalo:

```
import java.util.Scanner;

public class Principal {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce radio de la circunferencia: ");
        String num = lector.nextLine();
        lector.close();
        int radio;
        try {
            radio = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            System.out.println("Ha ocurrido una excepción de tipo
NumberFormatException");
            System.out.println(e.getMessage());
            return;
        }
        System.out.println("Longitud: " + (2*Math.PI*radio));
        System.out.println("Área: " + (Math.PI*radio*radio));
        System.out.println("Fin del programa");
    }
}
```

Introducimos por teclado el radio de una circunferencia como un texto, dentro de un bloque `try` lo convertimos a valor numérico. Al final del programa estamos calculando y mostrando en pantalla la longitud y el área de la circunferencia, pero como no tiene sentido que estas sentencias se ejecuten si hay error en la entrada de datos, hemos colocado un `return` al final del bloque `catch`, lo que provoca que termine la ejecución del programa si se desencadenó la excepción y no ejecuta el resto del código que queda en la función. Sin embargo, nos gustaría

que pase lo que pase aparezca el mensaje de "*Fin del programa*", y es aquí donde viene a salvarnos el bloque *finally*.

```
import java.util.Scanner;

public class Principal {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce radio de la circunferencia: ");
        String num = lector.nextLine();
        lector.close();
        int radio;
        try {
            radio = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            System.out.println("Ha ocurrido una excepción de tipo
NumberFormatException");
            System.out.println(e.getMessage());
            return;
        } finally {
            System.out.println("Fin del programa");
        }
        System.out.println("Longitud: " + (2*Math.PI*radio));
        System.out.println("Area: " + (Math.PI*radio*radio));
    }
}
```

El bloque *finally* se ejecuta siempre, independientemente del código que esté encerrado en cada uno de los *catch*.

¿Cómo se trabaja con las excepciones comprobadas?

Las excepciones comprobadas son las que pueden producir determinados métodos, cuyas llamadas deben estar obligatoriamente encerradas dentro de un bloque *try* o, en su defecto, estas excepciones también pueden ser relanzadas para que las gestione el método superior dentro de la cadena de llamadas.

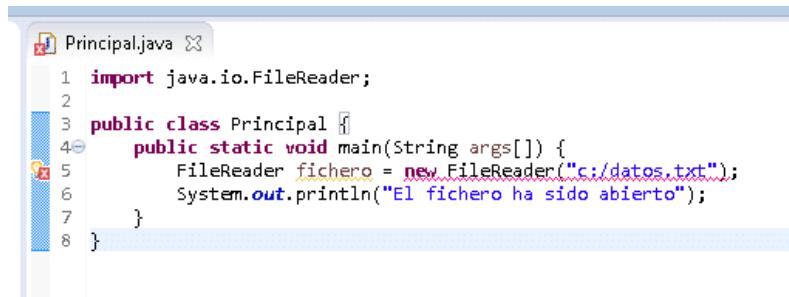
En nuestro ejemplo de la división entre 0 no estábamos obligados a poner un bloque *try*. El programa no tenía errores de compilación, aunque al ejecutar, nos lanzaba la excepción, pero hay determinadas situaciones en que estamos obligados a gestionar las situaciones de error. Vamos a ver un pequeño ejemplo para que puedas comprenderlo mejor.

Prueba a crear este pequeño programa dentro de Eclipse:

```
import java.io.FileReader;
```

```
public class Principal {
    public static void main(String args[]) {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Verás que ni siquiera has intentado ejecutar y ya estás teniendo problemas, Eclipse te está indicando que tienes errores de compilación, tu programa no puede ni siquiera ser ejecutado.



```
Principal.java
1 import java.io.FileReader;
2
3 public class Principal {
4     public static void main(String args[]) {
5         FileReader fichero = new FileReader("c:/datos.txt");
6         System.out.println("El fichero ha sido abierto");
7     }
8 }
```

El error de compilación viene porque el método constructor de *FileReader* abre un fichero para lectura cuya ruta y nombre se especifica como argumento. El fichero especificado podría no existir, y eso es algo que escapa a nuestro control, porque no podemos predecir si el usuario va a borrar el archivo o lo va a mover de sitio. Por esta razón, estamos obligados a controlar dicha excepción.

Para poder ejecutar el programa tienes que usar obligatoriamente un bloque *try*. Cambia ahora el código de esta manera:

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        try {
            FileReader fichero = new FileReader("c:/datos.txt");
            System.out.println("El fichero ha sido abierto");
        } catch (FileNotFoundException e) {
            System.out.println("Error al abrir el fichero");
            System.out.println(e.getMessage());
        }
    }
}
```

En resumen:

- **Excepciones comprobadas:** *try ... catch* obligatorio.
- **Excepciones no comprobadas:** *try ... catch* opcional.

Provocando excepciones no comprobadas

En este apartado tendrás oportunidad de provocar excepciones de tipo "no comprobadas". Como no es obligatorio usar *try ... catch* en este tipo de excepciones, no lo usaremos, solo vamos a familiarizarnos con más tipos de excepciones, a parte de la ya conocida *ArithmeticException*.

Ve probando con Eclipse los pequeños programas que te vamos a proponer. Ejecuta cada ejemplo para que veas con tus propios ojos lo que ocurre.

1. Salida de los límites de un array

```
public class Principal {
    public static void main(String args[]) {
        int nums[] = new int[3];
        nums[4]=25;
    }
}
```

Estamos accediendo a una posición inexistente del *array* y se produce la excepción *ArrayIndexOutOfBoundsException*.

2. Salida de los límites de una colección

```
import java.util.ArrayList;

public class Principal {
    public static void main(String args[]) {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        nums.add(25);
        nums.add(56);
        nums.add(18);
        System.out.println(nums.get(4));
    }
}
```

Estamos intentando recuperar un elemento de la colección de una posición inexistente. Se produce la excepción *IndexOutOfBoundsException*.

3. Error en la conversión entre tipos de datos

```
public class Principal {
    public static void main(String args[]) {
        String texto = "pepe";
        int num = Integer.parseInt(texto);
    }
}
```

Estamos intentando convertir el valor de tipo *String* "pepe" a formato numérico, cosa que no es posible y provoca una excepción de tipo *NumberFormatException*.

4. Intento de utilización de una referencia con valor *null*

```
public class Principal {
    public static void main(String args[]) {
        String texto=null;
        System.out.println(texto.toString());
    }
}
```

Estamos intentando utilizar una referencia a un *String* que en realidad no apunta a ningún objeto, tiene el valor *null*. Se produce una excepción de tipo *NullPointerException*.

Recuerda que aunque no tienes obligación de controlar estas excepciones con un *try ... catch*, aunque puedes hacerlo si lo consideras necesario.

Varios catch

En esta ocasión probaremos con un pequeño programa que puede producir dos tipos distintos de excepciones.

Este pequeño programa permite al usuario introducir por teclado los valores de un dividendo y un divisor, almacenándolos en variables tipo *String*. Luego, convierte ambos valores a datos numéricos de tipo *int* y por último calcula la división y el resto, mostrándolos en pantalla.

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Introduce dividendo: ");
        String texto = lector.nextLine();
        int dividendo = Integer.parseInt(texto);
        System.out.println("Introduce divisor: ");
        texto = lector.nextLine();
        int divisor = Integer.parseInt(texto);
        int resultado = dividendo/divisor;
        int resto = dividendo%divisor;
        System.out.println("Resultado división: " + resultado);
        System.out.println("Resto: " + resto);
        lector.close();
    }
}
```

Esto puede dar lugar a tres situaciones diferentes:

1. Que el usuario introduzca correctamente los dos valores y el programa funcione sin excepciones.

```
<terminated> Principal (16) [Java App]
Introduce dividendo:
16
Introduce divisor:
3
Resultado división: 5
Resto: 1
```

2. Que el usuario introduzca un valor de texto que no pueda ser convertido a número.

```
Introduce dividendo:
16
Introduce divisor:
pepe
Exception in thread "main" java.lang.NumberFormatException: For input string: "pepe"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at Principal.main(Principal.java:11)
```

3. Que el usuario introduzca 0 como divisor.

```
Introduce dividendo:  
16  
Introduce divisor:  
0  
Exception in thread "main" java.lang.ArithmetricException: / by zero  
at Principal.main(Principal.java:12)
```

¿Cómo podemos tener controladas todas estas situaciones?

Tras un bloque *try* pueden existir tantos bloques *catch* como sean necesarios:

```
import java.util.Scanner;  
  
public class Principal {  
    public static void main(String args[]) {  
        try {  
            Scanner lector = new Scanner(System.in);  
            System.out.println("Introduce dividendo: ");  
            String texto = lector.nextLine();  
            int dividendo = Integer.parseInt(texto);  
            System.out.println("Introduce divisor: ");  
            texto = lector.nextLine();  
            int divisor = Integer.parseInt(texto);  
            int resultado = dividendo/divisor;  
            int resto = dividendo%divisor;  
            System.out.println("Resultado división: " + resultado);  
            System.out.println("Resto: " + resto);  
            lector.close();  
        }  
        catch (ArithmetricException e1) {  
            System.out.println("Se ha producido una ArithmetricException");  
            System.out.println(e1.getMessage());  
        }  
        catch (NumberFormatException e2) {  
            System.out.println("Se ha producido un NumberFormatException");  
            System.out.println(e2.getMessage());  
        }  
        System.out.println("El programa sigue aquí, no se ha abortado");  
    }  
}
```

El objeto de excepción será recogido por el bloque *catch* cuyo tipo de argumento coincide con el tipo de excepción que se ha producido.

Cuando colocamos varios *catch* es importante tener en cuenta el orden en que se colocan. Deben ir situados en orden de las clases más específicas a las más genéricas, o lo que es lo mismo, desde las inferiores en el árbol jerárquico a las superiores.

Observa este código:

```
try {
    FileReader f = new FileReader("C:/datos.txt");
    int x = f.read();
    System.out.println(x);
    f.close();
} catch (FileNotFoundException e1) {
    System.out.println("El fichero no se encuentra");
} catch (IOException e2) {
    System.out.println(e2.getMessage());
} catch (Exception e3) {
    System.out.println(e3.getClass().getName());
    System.out.println(e3.getMessage());
}
```

Dentro del bloque *try* intentamos abrir un fichero para lectura y después el programa realiza alguna operación de lectura. No te preocupes por comprender el código encerrado en el *try*, lo único que nos ocupa ahora es la comprensión de las excepciones. Si el fichero no puede abrirse por cualquier circunstancia, irá recorriendo secuencialmente los bloques *catch* en busca de la primera referencia que pueda capturar la excepción que se ha producido. Si el primer bloque *catch* fuese el de *Exception*, hubiera capturado la excepción sea cual sea, ya que un objeto *FileNotFoundException* o *IOException* también es *Exception*.

Multicatch

Una de las novedades a partir de la versión 8 de Java son los llamados *multipath*, capaces de recoger con un solo bloque *catch* distintos tipos de excepciones. Para separar cada uno de los tipos de excepción se utiliza el carácter |.

El programa anterior utilizando *multipath* quedaría así:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (NumberFormatException | ArithmeticException e) {
            System.out.println("Se ha producido una excepción de tipo " +
e.getClass().getName());
            System.out.println(e.getMessage());
        }
        System.out.println("El programa sigue aquí, no se ha abortado");
    }
}
```

Utilizamos *e.getClass().getName()* para averiguar cuál de las dos excepciones ocurrió.

Polimorfismo y excepciones

Recuerda que todas las clases que representan excepciones heredan directa o indirectamente de la clase *Exception*.

Esto significa que los objetos de tipo *NumberFormatException*, *ArithmeticException*, *IndexOutOfBoundsException*, *FileNotFoundException*, etc., son a la vez objetos *Exception*. Esto nos permite aplicar las ventajas del polimorfismo, es decir, tratar todos estos objetos de manera genérica como excepciones.

Sea cual sea el tipo de excepción que produzca nuestro programa, podrá ser capturada de esta manera:

```
import java.util.Scanner;

public class Principal {
    public static void main(String args[]) {
        try {
            Scanner lector = new Scanner(System.in);
            System.out.println("Introduce dividendo: ");
            String texto = lector.nextLine();
            int dividendo = Integer.parseInt(texto);
            System.out.println("Introduce divisor: ");
            texto = lector.nextLine();
            int divisor = Integer.parseInt(texto);
            int resultado = dividendo/divisor;
            int resto = dividendo%divisor;
            System.out.println("Resultado división: " + resultado);
            System.out.println("Resto: " + resto);
            lector.close();
        }
        catch (Exception e) {
            System.out.println("Se ha producido una Excepción de tipo " +
e.getClass().getName());
            System.out.println(e.getMessage());
        }

        System.out.println("El programa sigue aquí, no se ha abortado");
    }
}
```

Una referencia de tipo *Exception* puede apuntar a cualquier tipo de objeto de excepción. Utilizamos *sg.getClassName()* para averiguar qué excepción ocurrió.

Aunque este sistema resulta muy cómodo, no hay que abusar de su uso, ya que evita el hecho de poder tratar cada tipo de excepción de manera más especializada y profesional.

Relanzamiento o propagación de excepciones

Aunque estamos obligados a controlar las excepciones de tipo "comprobadas", también podemos propagarlas o relanzarlas hacia arriba para que sean controladas en el método superior en la pila de llamadas.

Observa este ejemplo y prueba a añadirlo a un proyecto Eclipse:

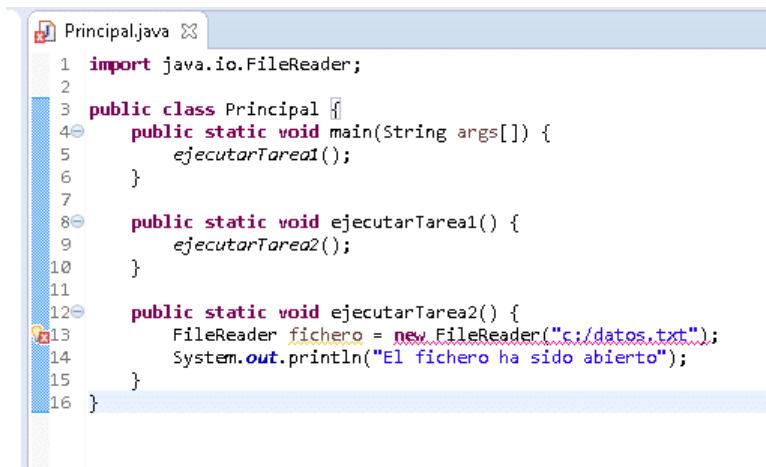
```
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

El método `ejecutarTarea2()` es invocado desde el método `ejecutarTarea1()`, que es llamado desde el método `main()`. Si ya has pegado el código en Eclipse estarás comprobando que de nuevo te da error de compilación porque es obligatorio controlar la posible excepción de tipo `FileNotFoundException` que puede provocar el constructor de `FileReader`. Deberíamos añadir el bloque `try ... catch` en el método `ejecutarTarea2()`.



Sin embargo, también **es posible que el método `ejecutarTarea2()` propague o relance la excepción hacia arriba** para responsabilizar al método superior. **Esto se hace con una declaración `throws` en la cabecera del método, indicando las excepciones que vayan a relanzarse.**

```

import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}

```

Propagación de la excepción desde `ejecutarTarea2()` hasta `ejecutarTarea1()`.

En este caso le hemos pasado la responsabilidad al método `ejecutarTarea1()`, que es el que ahora tiene el problema, tal y como puedes apreciar en la siguiente imagen:

```

Principal.java
1 import java.io.FileNotFoundException;
2 import java.io.FileReader;
3
4 public class Principal {
5     public static void main(String args[]) {
6         ejecutarTarea1();
7     }
8
9     public static void ejecutarTarea1() {
10        ejecutarTarea2();
11    }
12
13     public static void ejecutarTarea2() throws FileNotFoundException {
14         FileReader fichero = new FileReader("c:/datos.txt");
15         System.out.println("El fichero ha sido abierto");
16     }
17 }

```

Por otro lado, el método `ejecutarTarea1()` también podría "escurrir el bulto" pasando el problema al método `main()` de la misma forma:

```

import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}

```

Propagación de la excepción desde `ejecutarTarea1()` hasta `main()`.

Ahora es el método *main()* el que tiene el problema.

```

1 import java.io.FileNotFoundException;
2 import java.io.FileReader;
3
4 public class Principal {
5     public static void main(String args[]) {
6         ejecutarTarea1();
7     }
8
9     public static void ejecutarTarea1() throws FileNotFoundException {
10        ejecutarTarea2();
11    }
12
13     public static void ejecutarTarea2() throws FileNotFoundException {
14         FileReader fichero = new FileReader("c:/datos.txt");
15         System.out.println("El fichero ha sido abierto");
16     }
17 }
```

En el método *main()* podemos quitar los problemas de compilación de dos formas:

1. De nuevo propagando la excepción hacia arriba, pero como ya no hay más métodos para recogerla, será recibida por la máquina virtual de Java, que abortará el programa en el caso de que realmente se produzca la excepción, es decir, en el caso de que no exista el fichero que estamos intentando abrir.

```

import java.io.FileNotFoundException;
import java.io.FileReader;

public class Principal {
    public static void main(String args[]) throws FileNotFoundException {
        ejecutarTarea1();
    }

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

2. Colocar un *try ... catch*:

```

import java.io.FileNotFoundException;
import java.io.FileReader;

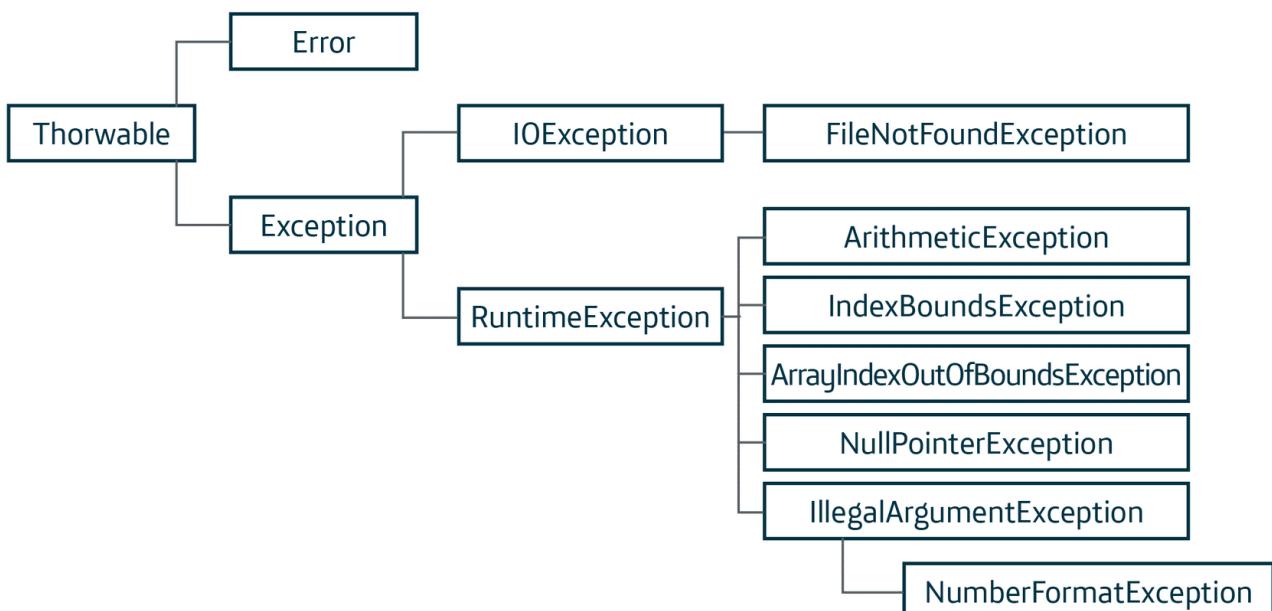
public class Principal {
    public static void main(String args[]) {
        try {
            ejecutarTarea1();
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
            return;
        }
    }

    public static void ejecutarTarea1() throws FileNotFoundException {
        ejecutarTarea2();
    }

    public static void ejecutarTarea2() throws FileNotFoundException {
        FileReader fichero = new FileReader("c:/datos.txt");
        System.out.println("El fichero ha sido abierto");
    }
}
```

Resumen de las excepciones más habituales

Pulsa sobre los círculos para obtener información adicional sobre las clases de excepción que hemos visto durante la unidad.



Throwable

Clase base de la que derivan todas las demás clases de excepción.

Error

Errores graves que escapan a nuestro control, tal como un fallo de la máquina virtual de Java.

Exception

Excepciones que sí podemos controlar y que se clasifican en:

- Comprobadas: las que derivan directamente de **Exception**. Es obligatorio gestionarlas.
- No comprobadas: las que derivan de **RuntimeException**. No es obligatorio gestionarlas.

IOException

Excepciones producidas durante operaciones de entrada y/o salida.

RuntimeException

Clase base de la que derivan todas las excepciones no comprobadas.

FileNotFoundException

Se está intentando acceder a un fichero que no existe.

ArithmaticException

Errores en operaciones aritméticas, como por ejemplo una división entre 0.

IndexOutOfBoundsException

Intento de acceso a una posición inexistente de una colección.

ArrayIndexOutOfBoundsException

Intento de acceso a un elemento de un *array* que existe, es decir, intento de salirse de los límites del *array*.

NullPointerException

Intento de acceder a una propiedad o a un método a partir de una referencia nula.

IllegalArgumentException

Se pasa un argumento a una función que resulta inválido para la operación que la función debe realizar.

NumberFormatException

Formato de número incorrecto. Ocurre cuando realizamos una conversión de texto a número y el texto no puede interpretarse como un número.

Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Las **clases son abstracciones de entidades en el mundo real** que sirven como modelos para construir objetos basados en ellas.
- Las clases **tienen atributos**. Los atributos pueden pertenecer al objeto (**propiedades del objeto**) o a la clase (**propiedades que pertenecen a la clase**). Los atributos que pertenecen a la clase se declaran con el modificador *static*.
- Los objetos **realizan acciones** que se implementan por medio de **métodos**. Los métodos pueden recibir argumentos y retornar un valor.
- **Los métodos pueden estar sobrecargados**, es decir, estar implementados de distintas maneras variando el número y/o tipo de parámetros.
- La interfaz de una clase está definida por las propiedades y métodos que dejamos ver al exterior.
- La **agregación y la composición son tipos de relaciones entre clases, donde una propiedad de una clase es un objeto de otra clase**.
- La **herencia** es una característica de la programación orientada a objetos que permite **crear una clase que hereda las propiedades y métodos de otra clase madre**.
- En el contexto de la programación orientada a objetos, **el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos y se puede lograr de varias formas**.
- Cuando se produce una **condición de error o excepción**, se crea un objeto que representa esa excepción y se envía al método en el que se ha producido el error para que lo maneje.
- **Todas las excepciones** que podemos capturar **heredan de la clase *Exception***.

5.2. Eventos y asociación de acciones a eventos



Índice

Objetivos	3
Eventos, fuentes y auditores de eventos	4
Eventos	4
Tipos de eventos. Mecanismo de gestión de eventos	5
Mecanismo de gestión de eventos	5
Gestionar eventos relacionados con las ventanas	7
Responder al evento clic de los botones	12
Los objetos Event	13
Librerías de clases asociadas	14
Librerías asociadas a la gestión de eventos	14
Despedida	17
Resumen.....	17

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Comprender el concepto de evento.
- Distinguir entre fuente y auditor de eventos.
- Crear aplicaciones utilizando mecanismos de gestión de eventos.

Eventos, fuentes y auditores de eventos

Eventos

Un evento es algo así como un suceso relevante ocurrido con un objeto y al que queremos responder.

Los objetos **se disparan** cuando ocurre algo que merece atención especial y deseamos que el resto de la aplicación escuche y pueda responder en consecuencia.

El objeto que dispara el evento es el **objeto fuente**.

Los objetos que escuchan el evento son los **objetos auditores**.

Ejemplos:

- Una clase *Cuenta*, que representa una cuenta bancaria, podría disparar un evento cuando la cuenta se queda en números rojos. Otra clase *Cliente*, que construye objetos de tipo *Cuenta*, puede auditar o escuchar dichos eventos para responder en consecuencia.
- En una clase que representa una ventana en un programa de escritorio, sus objetos podrán disparar un evento cuando el usuario hace clic en un botón. Otra clase *Cliente* que construye un objeto para añadir dicha ventana a su aplicación, podrá responder a dicho evento para asociar alguna acción al hecho de hacer clic en el botón.

Tipos de eventos. Mecanismo de gestión de eventos

Mecanismo de gestión de eventos

Antes de nada, para que puedas comprender mejor los mecanismos de gestión de eventos, es conveniente que crees un proyecto nuevo en Eclipse con una clase que representará una ventana de una aplicación de escritorio.

No es objetivo de este curso profundizar en las aplicaciones de escritorio, solo veremos lo necesario para poder aplicarlo a la gestión de eventos.

Para nuestro ejemplo utilizaremos la **librería de clases javax.swing**.

¡Manos a la obra!

1. Crea un proyecto en eclipse llamado **ProyectoEventos**.

2. Crea una **clase** llamada **Ventana** con el siguiente código:

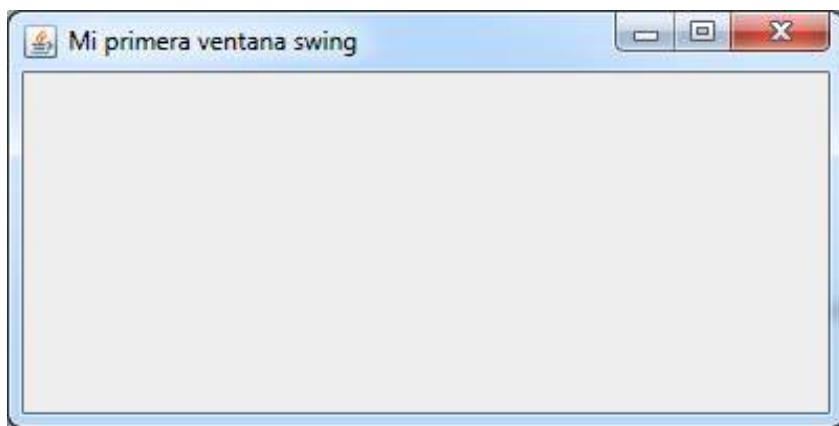
```
import javax.swing.*;  
  
public class Ventana extends JFrame {  
    private static final long serialVersionUID = 1L;  
  
    Ventana() {  
        super ("Mi primera ventana swing");  
    }  
  
    public void iniciar() {  
        this.setLocation(200,200);  
        this.setSize(400,200);  
        this.setVisible(true);  
    }  
}
```

3. Crea una **clase** llamada **Principal** con el siguiente código:

```
public class Principal {  
    public static void main(String args[]) {  
        Ventana v = new Ventana();  
        v.iniciar();  
    }  
}
```

4. Ejecuta desde la clase *Principal*.

Como resultado te aparecerá una ventana como esta:

**Parece magia, ¿verdad?**

Aunque no sepas nada sobre la librería **javax.swing**, sí sabes ya mucho sobre programación orientada a objetos en Java y seguro que no te resulta complicado.

En la clase *Principal* no estamos haciendo nada que no hayamos hecho ya. Utilizamos el método *main* para construir un objeto de la clase *Ventana* y llamamos a un método, el método *iniciar()*.

La clase *Ventana* extiende o hereda de la clase **JFrame**.

```
public class Ventana extends JFrame {  
}
```

Y de *JFrame* ha heredado el aspecto de ventana, no sabemos cómo ni nos interesa. Lo importante es que podemos construir una aplicación con ventanas.

En el constructor de *Ventana* tenemos que pasar un argumento al constructor de *JFrame*. *JFrame* utiliza este argumento para poner título a la ventana.

```
Ventana() {  
    super ("Mi primera ventana swing");  
}
```

Con el método *iniciar()* configuramos detalles de la ventana, en este caso la posición, alto, ancho y visibilidad.

¿Pero para qué queremos una ventana vacía? Para nada, hay que añadirle elementos de tipo interfaz de usuario, botones, cajas de texto, listas desplegables, menús, etc.

Puesto que nuestra clase *Ventana* tiene un método *iniciar()* que la configura, será aquí donde podamos añadir elementos que antes habrá que definir.

Modifica la clase *Ventana* de la siguiente manera:

```
public class Ventana extends JFrame {  
    private static final long serialVersionUID = 1L;  
    private JPanel p=new JPanel();  
    private JButton botonAzul = new JButton("Azul");  
    private JButton botonVerde = new JButton("Verde");  
    private JButton botonRojo = new JButton("Rojo");  
  
    Ventana() {  
        super ("Mi primera ventana swing");  
    }  
  
    public void iniciar() {  
        this.setLocation(200,200);  
        this.setSize(400,200);  
        this.setVisible(true);  
        this.add(p); // Añadimos panel a la ventana.  
        p.add(botonRojo);  
        p.add(botonAzul);  
        p.add(botonVerde);  
    }  
}
```

Ahora tienes tres botones, pero al hacer clic en ellos no pasa nada. Aquí es donde entran en juego los mecanismos de gestión de eventos.

Los objetos de la clase ***JButton*** “disparan” un evento cuando el usuario hace clic en el botón, pero la clase *Ventana* (cliente o usuaria de la clase *JButton*) no está “escuchando o auditando” dicho evento.

El modelo de eventos en Java se denomina gestión de eventos delegado. Se debe registrar específicamente si se quiere gestionar un evento, como puede ser hacer clic sobre un botón. De esta forma, Java mejora el rendimiento de las aplicaciones. Si solo “se disparan” los eventos que necesitamos estaremos ahorrando recursos necesarios.

Los eventos se registran implementando una interfaz de *Listener* de eventos que necesitemos.

Gestionar eventos relacionados con las ventanas

Vamos a comenzar por implementar la interfaz de *Listener* asociada con la gestión de eventos de la ventana (cargar ventana, cerrar ventana, iconizar, activar, etc.). En otro apartado pasaremos a gestionar el clic de los botones.

Tu clase *Ventana*, además de extender de *JFrame*, debe implementar la ***interface WindowListener***.

```
public class Ventana extends JFrame implements WindowListener {  
}
```

Ahora mismo tendrás un error de compilación y la palabra *WindowListener* está subrayada en rojo. Es porque tienes que importar la librería de clases que contiene la clase *WindowListener*.

Si sitúas el puntero de ratón sobre la palabra subrayada *WindowListener* obtendrás una pequeña ventana flotante que te da pistas sobre la solución al error de compilación. En concreto la solución está en importar la librería ***java.awt.event*** que contiene todas las clases relacionadas con la gestión de eventos en ventanas.

The screenshot shows a Java code editor with the following code:

```
1 import javax.swing.*;  
2  
3 public class Ventana extends JFrame implements WindowListener {  
4     private static final long serialVersionUID = 1L;  
5     private JPanel p=new JPanel();  
6     private JButton botonAzul = new JButton("A");  
7     private JButton botonVerde = new JButton("V");  
8     private JButton botonRojo = new JButton("R");  
9  
10    Ventana() {  
11        super ("Mi primera ventana swing");  
12    }  
13  
14    public void iniciar() {  
15        this.setLocation(200,200);  
16        this.setSize(400,200);  
17        this.setVisible(true);  
18        this.add(p); // Añadimos panel a la ventana
```

A tooltip window is displayed over the word *WindowListener*, which says: "WindowListener cannot be resolved to a type". It lists 6 quick fixes available, with the first one circled in red: "Import 'WindowListener' (java.awt.event)".

Eclipse te ha añadido el *import* que faltaba, has solucionado un error, pero te ha aparecido otro, **ahora está subrayada en rojo la palabra *Ventana***.

Recuerda que **una clase que implementa una interfaz está obligada a implementar los métodos abstractos de dicha interfaz**. Por esa razón ahora tienes un error de compilación en la clase *Ventana*.

De nuevo puedes situar el puntero de ratón, pero ahora sobre la palabra *Ventana* para ver qué solución te da Eclipse.

The screenshot shows the same Java code as before, but now there is a warning message above the class definition: "The type Ventana must implement the inherited abstract method WindowListener.windowDeactivated(WindowEvent)". A tooltip window is shown over the word *Ventana*, with the first option circled in red: "Add unimplemented methods".

```
1 import java.awt.event.WindowListener;  
2 import javax.swing.*;  
3  
4 public class Ventana extends JFrame implements WindowListener {  
5     private JPanel p;  
6     private JButton botonAzul = new JButton("A");  
7     private JButton botonVerde = new JButton("V");  
8     private JButton botonRojo = new JButton("R");  
9  
10    Ventana() {  
11        super ("Mi primera ventana swing");  
12    }  
13  
14    public void iniciar() {  
15        this.setLocation(200,200);  
16        this.setSize(400,200);  
17        this.setVisible(true);  
18        this.add(p); // Añadimos panel a la ventana
```

Ahora la solución está en implementar los métodos de la interfaz *WindowListener* (**Add unimplemented methods**).

Si lo has hecho ya, de manera automática se han generado los siguientes métodos:

```
@Override  
public void windowActivated(WindowEvent arg0) {  
  
}  
  
@Override  
public void windowClosed(WindowEvent arg0) {  
  
}  
  
@Override  
public void windowClosing(WindowEvent arg0) {  
  
}  
  
@Override  
public void windowDeactivated(WindowEvent arg0) {  
  
}  
  
@Override  
public void windowDeiconified(WindowEvent arg0) {  
  
}  
  
@Override  
public void windowIconified(WindowEvent arg0) {  
  
}  
  
@Override  
public void windowOpened(WindowEvent arg0) {  
  
}
```

Ahora puedes añadir algo de código a estos métodos manejadores de evento, solo como prueba:

```
@Override  
public void windowActivated(WindowEvent arg0) {  
    System.out.println("Se ha activado la ventana");  
}  
  
@Override  
public void windowClosed(WindowEvent arg0) {  
    System.out.println("Se ha terminado de cerrar la ventana");  
}  
  
@Override  
public void windowClosing(WindowEvent arg0) {  
    System.out.println("Se está comenzando a cerrar la ventana");  
}  
@Override  
public void windowDeactivated(WindowEvent arg0) {  
    System.out.println("Se está desactivando la ventana");  
}
```

```
@Override  
public void windowDeiconified(WindowEvent arg0) {  
    System.out.println("Se está restaurando una ventana que estaba iconizada");  
}  
  
@Override  
public void windowIconified(WindowEvent arg0) {  
    System.out.println("Se está iconizando la ventana");  
}  
  
@Override  
public void windowOpened(WindowEvent arg0) {  
    System.out.println("Se está abriendo la ventana");  
}
```

Si ejecutas de nuevo, deberías estar viendo mensajes en la consola de Eclipse, pero,

¡Oh! ¡No sale ningún mensaje! ¿No funcionan los eventos?

No te preocupes, es que nos falta un paso todavía.

Recuerda que el modelo de eventos en Java se denomina gestión de eventos delegado. Se debe registrar específicamente si se quiere gestionar un evento. Justo eso es lo que nos falta. No hemos especificado en ningún momento que queremos gestionar los eventos relacionados con la ventana. Y debemos hacerlo así:

```
this.addWindowListener(this);
```

Para cada objeto cuyos eventos deseamos gestionar, hay que utilizar un método que siempre comenzará por *add*.

En este caso, el objeto es la propia ventana, es decir, el objeto actual (*this*).

Ahora nuestra clase ventana está así:

```
import java.awt.event.WindowEvent;  
import java.awt.event.WindowListener;  
import javax.swing.*;  
  
public class Ventana extends JFrame implements WindowListener {  
    private static final long serialVersionUID = 1L;  
    private JPanel p=new JPanel();  
    private JButton botonAzul = new JButton("Azul");  
    private JButton botonVerde = new JButton("Verde");  
    private JButton botonRojo = new JButton("Rojo");  
  
    Ventana() {  
        super ("Mi primera ventana swing");  
    }  
  
    public void iniciar() {  
        this.setLocation(200,200);  
        this.setSize(400,200);  
        this.setVisible(true);  
        this.add(p); // Añadimos panel a la ventana.
```

```
p.add(botonRojo);
p.add(botonAzul);
p.add(botonVerde);
this.addWindowListener(this);
}

@Override
public void windowActivated(WindowEvent arg0) {
    System.out.println("Se ha activado la ventana");
}

@Override
public void windowClosed(WindowEvent arg0) {
    System.out.println("Se ha terminado de cerrar la ventana");
}

@Override
public void windowClosing(WindowEvent arg0) {
    System.out.println("Se está comenzando a cerrar la ventana");
}

@Override
public void windowDeactivated(WindowEvent arg0) {
    System.out.println("Se está desactivando la ventana");
}

@Override
public void windowIconified(WindowEvent arg0) {
    System.out.println("Se está restaurando una ventana
que estaba iconizada");
}

@Override
public void windowIconified(WindowEvent arg0) {
    System.out.println("Se está iconizando la ventana");
}

@Override
public void windowOpened(WindowEvent arg0) {
    System.out.println("Se está abriendo la ventana");
}
}
```

Ahora, si ejecutas el proyecto, irás viendo un montón de mensajes en la consola de Eclipse mientras vas interactuando con la ventana.

Resumiendo, para gestionar los eventos relacionados con las acciones de la ventana, hemos tenido que seguir estos pasos:

- Implementar la interfaz de *listener WindowListener* añadiendo “**implements WindowListener**” a la cabecera de la clase *Ventana*.
- **Implementar los métodos abstractos** de *WindowListener* (**windowActivated**, **windowClosed**, **windowClosing**, etc.).
- **Registrar** específicamente que deseamos gestionar los eventos relacionados con la ventana añadiendo la instrucción “**this.addWindowListener(this);**”

Responder al evento clic de los botones

Ahora vamos a responder al evento clic de los botones. Para conseguirlo vamos de nuevo a realizar tres pasos similares a los anteriores con los eventos de ventana. Veamos los pasos.

1. Implementa la interfaz de *listener ActionListener*

La interfaz **ActionListener** está pensada para gestionar los eventos de acción, tal como hacer clic sobre un botón.

```
public class Ventana extends JFrame implements WindowListener, ActionListener{  
}
```

2. Implementa los **métodos abstractos**

Implementa el único método abstracto de *ActionListener* (**actionPerformed**) añadiendo las acciones que quieras que ocurran al hacer clic en cada botón.

```
@Override  
    public void actionPerformed(ActionEvent e) {  
        if (e.getSource() == this.botonAzul)  
            this.p.setBackground(Color.BLUE);  
        else if (e.getSource() == this.botonRojo)  
            this.p.setBackground(Color.RED);  
        else  
            this.p.setBackground(Color.GREEN);  
    }
```

3. Indica los eventos que quieras gestionar

Registra específicamente que deseas gestionar los eventos relacionados con el clic de los botones. Para ello ejecuta el método **addActionListener()** sobre los objetos deseados. El método *iniciar()* quedará así:

```
public void iniciar() {  
    this.setLocation(200,200);  
    this.setSize(400,200);  
    this.setVisible(true);  
    this.add(p); // Añadimos panel a la ventana.  
    p.add(botonRojo);  
    p.add(botonAzul);  
    p.add(botonVerde);  
    this.addWindowListener(this);  
    this.botonAzul.addActionListener(this);  
    this.botonRojo.addActionListener(this);  
    this.botonVerde.addActionListener(this);  
}
```

Los objetos Event

A estas alturas ya habrás comprobado que los métodos manejadores de eventos como *actionPerformed()*, *windowIconified()*, *windowActivated()*, etc. llevan un parámetro. Este parámetro trae información relevante sobre el evento ocurrido.

Vamos a analizar de nuevo el código del método sobrescrito ***actionPerformed()*** en la clase *Ventana*.

```
@Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == this.botonAzul)
            this.p.setBackground(Color.BLUE);
        else if (e.getSource() == this.botonRojo)
            this.p.setBackground(Color.RED);
        else
            this.p.setBackground(Color.GREEN);
    }
```

El **objeto e de tipo ActionEvent** es un argumento que trae información sobre el evento ocurrido y quién lo provocó.

Concretamente, estamos utilizando el método ***getSource()***, que nos devuelve el objeto que “disparó” el evento; para nuestro ejemplo solo puede ser uno de estos tres objetos: *botonAzul*, *botonRojo* o *botonVerde*.

Librerías de clases asociadas

Librerías asociadas a la gestión de eventos

Las clases implicadas en la gestión de eventos con ventanas se encuentran encerradas en el paquete **java.awt.event**.

Recuerda que para registrar eventos en una clase primero hay que implementar la interfaz de *Listener* correspondiente. Estas son las interfaces de *Listener* disponibles:

ActionListener

Para registrar eventos de acción, como hacer clic sobre los botones.

AdjustmentListener

Para gestionar eventos en los que un componente es escondido, movido, redimensionado o mostrado.

ContainerListener

Para registrar eventos relacionados con que un componente coge o pierde el foco.

ItemListener

Para gestionar eventos relacionados con el cambio de elemento seleccionado en una lista.

KeyListener

Para gestionar los eventos relacionados con el teclado.

MouseListener

Para gestionar los eventos relacionados con el ratón.

MouseMotionListener

Para gestionar los eventos relacionados con el movimiento del ratón.

TextListener

Para gestionar los eventos de cambio de valor de texto.

WindowListener

Para gestionar los casos en que una ventana está activada, desactivada, con o sin forma de ícono, abierta, cerrada o se sale de ella.

Todas estas interfaces, heredan de la interface genérica **EventListener**, situada en el paquete **java.util**.

Cualquier aplicación puede crear sus propios manejadores de eventos creando interfaces personalizadas que hereden de `java.util.EventListener`.

Recuerda que cada *Listener* es una interfaz, y las clases que la implementan deben también implementar sus métodos abstractos. Cada uno de estos métodos tiene un parámetro que recibe información sobre el evento. Recuerda la estructura para el caso de **ActionListener**:

```
public class Ventana extends JFrame implements ActionListener {  
  
    // Método sobrescrito de ActionListener  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        // ActionEvent e es el parámetro que recibe con información  
        // sobre el evento.  
    }  
}
```

Los parámetros que reciben los métodos gestores de eventos en las aplicaciones con ventanas serán uno de los siguientes:

ActionEvent

Hacer clic en un botón, elemento de lista u otro componente.

AdjustmentEvent

Movimientos en la barra de desplazamiento.

ComponentEvent

Un componente es escondido, movido, ocultado o mostrado.

FocusEvent

Un componente coge o pierde el foco (cursor).

ItemEvent

Hacer clic en un elemento de la lista o en una casilla de verificación de un grupo.

KeyEvent

Acciones con el teclado.

MouseEvent

Acciones relacionadas con el ratón.

TextEvent

Cambios en cuadros de texto.

WindowListener

Acciones relacionadas con la ventana.

Todas estas clases son derivadas de la clase *EventObject* situada en el paquete *java.util*.

Cualquier aplicación puede crear sus propias clases gestoras de eventos personalizadas extendiendo de *java.util.EventObject*.

Despedida

Resumen

Has finalizado esta lección, veamos los puntos más importantes que hemos tratado.

La gestión de eventos en una clase Java consta de estos tres pasos:

1. Implementar la interfaz de *Listener* que necesitemos.

```
public class Ventana extends JFrame implements ActionListener {  
}
```

2. Implementar los métodos abstractos de la interfaz elegida.

```
public class Ventana extends JFrame implements ActionListener {  
    ...  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        // Acciones  
    }  
}
```

3. Registrar específicamente los eventos que queremos controlar en cada componente.

```
public void iniciar() {  
    ....  
    this.botonAzul.addActionListener(this);  
    this.botonRojo.addActionListener(this);  
    this.botonVerde.addActionListener(this);  
}
```

5.3. Empaquetado de componentes



Índice

Objetivos	3
Creación de librerías personalizadas	4
Crear una librería Java	4
Usar una librería Java	7
Despedida	9
Resumen.....	9

Objetivos

El contenido de esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.

Con esta lección perseguimos los siguientes objetivos:

- Construir librerías de clases Java empaquetadas en archivos .jar.
- Utilizar las librerías de clases en múltiples proyectos.

Creación de librerías personalizadas

Crear una librería Java

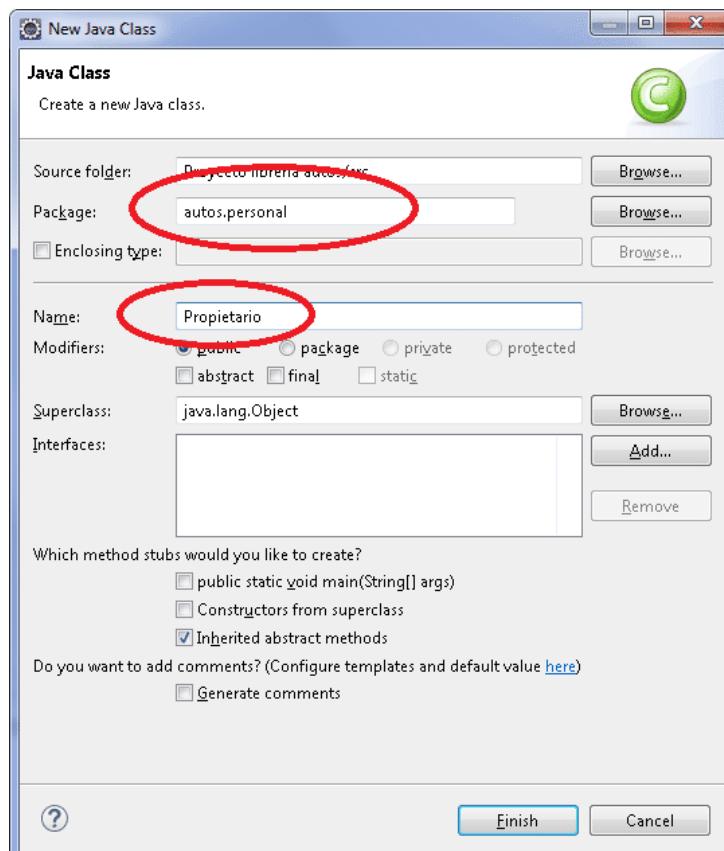
En esta lección recordarás cómo crear una librería de clases Java empaquetada en un archivo .jar para utilizarla después en múltiples proyectos.

En esta ocasión implementaremos las clases *Propietario* y *Coche*, y las empaquetaremos en una librería de clases situada en un archivo llamado *autos.jar*. Recuerda que las clases deberán estar organizadas en paquetes por una cuestión de orden.

Sigue estos pasos para crear la librería:

1. Crea un proyecto Java estándar llamado “Proyecto librería autos” (**File / New / Java Project**).
2. Ahora crea la clase *Propietario* dentro del paquete *autos.personal*. Para ello, haz **clic derecho sobre el nombre del proyecto** y selecciona en el menú contextual **New / Class**.

En el cuadro de diálogo “New Java Class” puedes especificar el nombre del paquete y el nombre de la clase, como ves en la imagen:



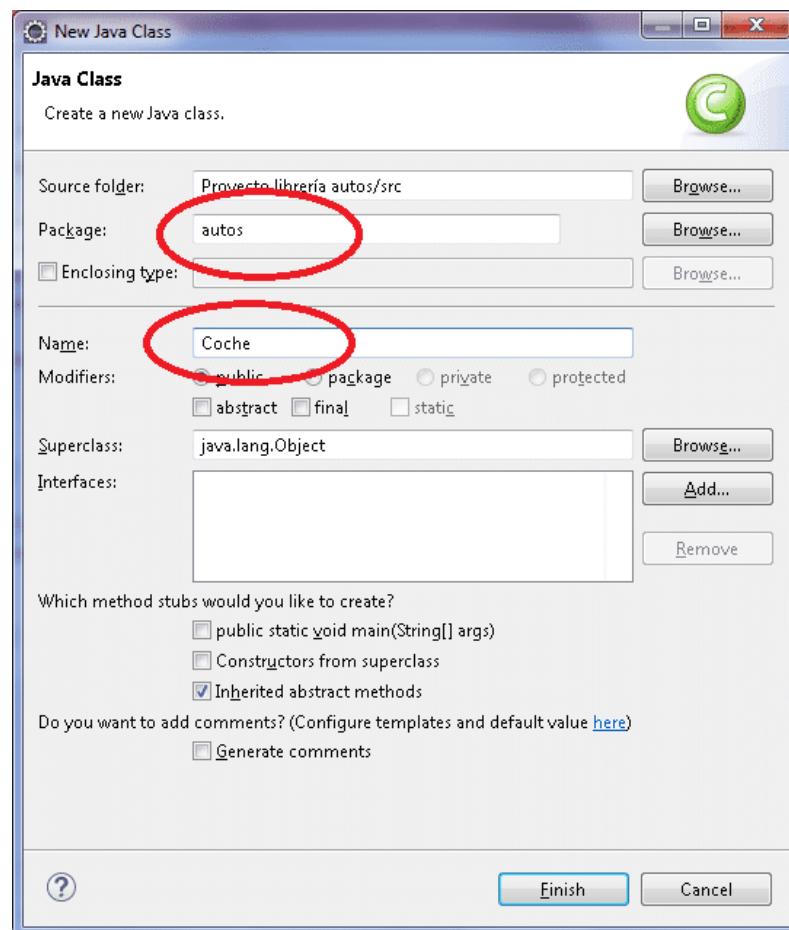
De manera automática se ha creado este código:

```
package autos.personal;  
  
public class Propietario {  
  
}
```

3. Completa el código de la clase *Propietario* dejándolo así:

```
package autos.personal;  
  
public class Propietario {  
    private String dni;  
    private String nombre;  
    private String apellidos;  
  
    public Propietario  
        (String dni, String nombre, String apellidos) {  
            this.dni = dni;  
            this.nombre = nombre;  
            this.apellidos = apellidos;  
    }  
  
    @Override  
    public String toString() {  
        return this.nombre + " " + this.apellidos +  
            " con DNI " + this.dni;  
    }  
}
```

4. Ahora crea la clase *Coche* dentro del paquete *autos*. Recuerda que debes hacer clic derecho sobre el nombre del proyecto y seleccionar en el menú contextual **New / Class**. Completa el cuadro de diálogo “New Java Class” de la siguiente manera:



De forma automática has conseguido este código:

```
package autos;

public class Coche {
```

5. Ahora completa el código de la clase *Coche* para dejarlo así:

```
package autos;

import autos.personal.Propietario;

public class Coche {
    private Propietario propietario;
    private String matricula;
    private String marca;
    private String modelo;
    private int velocidad;

    public Coche(String matricula, String marca,
                String modelo, Propietario propietario) {
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
```

```
        this.propietario = propietario;
        this.velocidad = 0;
    }

    public void acelerar(int cuanto) {
        this.velocidad = this.velocidad + cuanto;
    }

    public void frenar(int cuanto) {
        this.velocidad = this.velocidad - cuanto;
    }

    @Override
    public String toString() {
        return "El " + this.marca + " " + this.modelo +
            " con matricula " + this.matricula +
            " propiedad de " +
            this.propietario.toString() +
            " va a " + this.velocidad + " km/hora";
    }
}
```

6. Has llegado al punto clave, **crear el archivo .jar** que empaquetará el modelo de clases. El siguiente vídeo te muestra los pasos que debes seguir desde Eclipse para crear el archivo *.jar* en la ubicación deseada. <https://vimeo.com/telefonicaed/review/258245213/21ad3a7de1>

Usar una librería Java

Una vez que tengas creada tu librería de clases *autos.jar*, podrás utilizarla en múltiples proyectos.

Sigue estos pasos para ponerlo en práctica:

1. Crea un nuevo proyecto estándar Java llamado “Uso librería autos” (**File / New / Java Project**).

2. Entra en las propiedades del proyecto y **agrega la librería *autos.jar***. El siguiente vídeo te muestra los pasos necesarios.

<https://vimeo.com/telefonicaed/review/258245220/77e1e91fb6>

3. Ahora que has incluido la librería en tu proyecto, puedes actuar igual que si las clases *Propietario* y *Coche* estuvieran creadas directamente en este proyecto. Crea la clase *Principal* con el siguiente código:

```
import autos.Coche;
import autos.personal.Propietario;

public class Principal {
    public static void main(String[] args) {
        Propietario yo = new
            Propietario("66666666K", "Perico", "De los Palotes");
```

```
        Coche miCoche = new Coche("5577FJK", "Suzuki", "Ignis", yo);
        miCoche.acelerar(100);
        miCoche.frenar(10);
        System.out.println(miCoche.toString());
    }
}
```

Recuerda que se puede utilizar el carácter de asterisco como comodín para importar todas las clases situadas en un paquete.

```
import autos.*;
import autos.personal.*;
```

Así importamos todas las clases incluidas en *autos* y todas las clases incluidas en *personal*. Como en nuestro ejemplo sólo tenemos una clase en cada paquete, no hay diferencia entre este sistema y el usado en el ejemplo.

Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Una **librería** contiene un conjunto de clases que pueden servir a múltiples proyectos.
- **Las librerías de clases Java están organizadas en paquetes**, siendo un paquete una carpeta en el sistema de archivos.
- **Las librerías están empaquetadas en archivos .jar** que cada proyecto que la requiera deberá importar.

5.4. Polimorfismo, introspección y reflexión



Índice

Objetivos	3
Polimorfismo.....	4
Raíz etimológica de polimorfismo	4
Concepto de polimorfismo aplicado a la POO.....	4
Tipos de polimorfismo.....	4
Polimorfismo y relación de herencia.....	5
Polimorfismo: clases abstractas e interfaces.....	6
Las clases abstractas	6
Reglas de diseño de las clases abstractas:.....	6
Reglas para las clases que heredan una clase abstracta:	8
Conversión o cash entre clases	8
Las interfaces	11
Las interfaces y la herencia múltiple	13
Distinguiendo los distintos tipos de polimorfismo	14
Polimorfismo en tiempo de compilación (sobrecarga)	14
Ligadura dinámica: referencias polimórficas	15
Clases genéricas	18
Comprobación de la clase a la que pertenece un objeto	20
Comprobación dinámica de tipos.....	20
Introspección y reflexión.....	20
Despedida	22
Resumen.....	22

Objetivos

El contenido de esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.

Con esta lección perseguimos los siguientes objetivos:

- Comprender los conceptos de introspección y reflexión.
- Recordar el concepto de polimorfismo y su aplicación en la programación orientada a objetos, ya que tiene una relación directa con los conceptos de introspección y reflexión.
- Averiguar en tiempo de ejecución la clase a la que pertenece un objeto, algo muy útil cuando utilizamos polimorfismo.

Polimorfismo

Raíz etimológica de polimorfismo

En cuanto a la raíz etimológica de la palabra polimorfismo, es de origen griego y significa múltiples formas.

Sus componentes léxicos son:

- **Polys:** muchos.
- **Morfo:** formas.
- **Sufijo -ismo:** actividad o sistema.

En cuanto al polimorfismo como propiedad se aplica a todo ser vivo u objeto capaz de adoptar múltiples formas o capaz de pasar por numerosos estados.

La comprensión del concepto general de polimorfismo te ayudará a asimilar mejor su aplicación en el mundo de la programación.

Concepto de polimorfismo aplicado a la POO

En el contexto de la programación orientada a objetos el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos. Se puede lograr de varias formas.

Tipos de polimorfismo

Polimorfismo en tiempo de compilación (sobrecarga)

Capacidad de un método para adaptar distintos comportamientos en función de los parámetros recibidos. El mismo método se comporta de distintas formas.

Polimorfismo en tiempo de ejecución (ligadura dinámica)

Capacidad de una referencia a un objeto para apuntar a distintos tipos de objetos. Una referencia a un objeto de tipo *Figura* puede apuntar a objetos de tipo *Triangulo*, *Rectangulo* o *Circulo*.

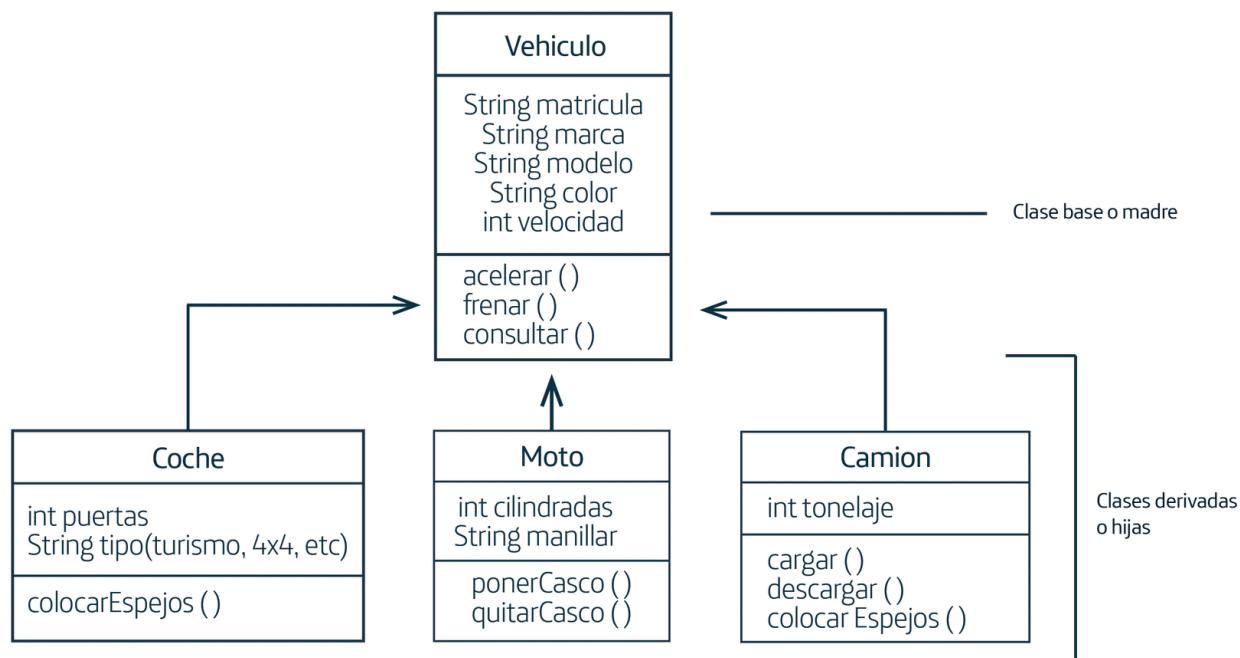
Clases genéricas

Una propiedad de una clase genérica puede variar de tipo según las necesidades. Por ejemplo: una clase llamada *Sumador* podría recibir dos argumentos que podrían ser números enteros (en cuyo caso halla la suma), cadenas de caracteres (en cuyo caso concatena ambas cadenas), dos arrays numéricos (en cuyo caso genera un nuevo array que suma los elementos que ocupan la misma posición de cada array).

Polimorfismo y relación de herencia

El polimorfismo se logra principalmente a través de relaciones de herencia.

Observa la siguiente imagen; una referencia a un objeto de tipo *Vehiculo* puede adoptar forma de objeto de tipo *Coche*, *Moto* o *Camion*.



El hecho de poder tratar un *Coche*, una *Moto* y un *Camion* de la misma forma (como un *Vehiculo*) favorece enormemente la reutilización de código. Lo descubrirás con los ejemplos que vayamos desarrollando.

Polimorfismo: clases abstractas e interfaces

Las clases abstractas

Una clase abstracta se diseña específicamente para que sirva de base a otras clases derivadas que la hereden, pero no sirve para crear objetos a partir de ella.

Están pensadas para crear otras clases derivadas que implementen sus métodos abstractos.

Ahora vamos a partir de un ejemplo de abstracción; la clase *Animal* con sus derivadas *Pulga* y *Tiranosaurio*.

Crea un nuevo proyecto y comienza por crear la clase abstracta *Animal*.

```
/*
 * Esta es una clase abstracta.
 * Sólo puede ser usada para crear clases derivadas
 */

public abstract class Animal {
    // Variable de instancia de clase.
    String nombre;

    // Constructor.
    public Animal(String n) {
        nombre = n;
    }

    // Métodos abstractos, deben ser sobre escritos en una clase derivada.
    public abstract String morder(Animal ani);
    public abstract String mover();

    // Método no abstracto, podrá ser o no sobre escrito en la clase derivada.
    @Override
    public String toString() {
        return "Saludos desde Animal";
    }
}
```

Reglas de diseño de las clases abstractas:

- Una clase abstracta se define con el modificador *abstract* (*abstract class ...*).
- No es posible crear objetos a partir de una clase abstracta (*new Animal(...)*).
- Las clases abstractas tienen como objetivo servir de base a otras clases derivadas.
- Una clase abstracta debe tener algún método abstracto.
- Los métodos abstractos no tienen implementación, solo se indica la cabecera del método. Estos métodos deberán ser implementados por las clases derivadas.
- Las clases abstractas también pueden tener métodos no abstractos (ya implementados). Pensando en el ejemplo de las figuras, la clase *Figura* podría

- implementar un método llamado *modificarCoordenadas()*, pero un método como *calculaArea()*, por lógica, debería ser abstracto.
- Una clase abstracta puede heredar de otra clase abstracta.

Ahora crea las clases derivadas de la clase abstracta *Animal* (*Pulga* y *Tiranosauro*):

```
/*
Esta es una clase derivada que hereda una clase
abstracta. Será obligatorio implementar los métodos
abstractos de la clase base.

*/
public class Pulga extends Animal {
    Pulga() {
        super ("Pulga");
    }

    @Override
    public String morder(Animal ani) {
        return "Pulga muerde "+ani.nombre;
    }

    @Override
    public String mover() {
        return "Pulga se mueve";
    }

    @Override
    public String toString() {
        return "Saludos de la pulga";
    }
}
```

```
public class Tiranosaurio extends Animal {
    public Tiranosaurio() {
        super ("Tiranosauro");
    }

    @Override
    public String morder(Animal ani) {
        return "Tiranosauro muerde "+ani.nombre;
    }

    @Override
    public String mover() {
        return "Tiranosauro se mueve";
    }

    @Override
    public String toString() {
        return "Saludos del tiranosauro";
    }

    // Esté método es exclusivo del tiranosauro
    public String pisar(Animal ani) {
        return "Tiranosauro pisa a "+ani.nombre;
    }
}
```

Recuerda: la anotación **@Override** indica que el método ha sido heredado y lo estamos sobrescribiendo.

Reglas para las clases que heredan una clase abstracta:

- Las clases derivadas de una clase abstracta están obligadas a sobrescribir sus métodos abstractos.
- En cuanto a los métodos no abstractos, podrán optar por sobrescribirlos o utilizarlos tal cual están originariamente.

Por último, puedes crear la clase *Principal* con método *main* para poner en práctica el uso de la estructura de clases.

```
public class Principal {  
    public static void main (String[] args) {  
        Animal anil = new Pulga();  
        Animal ani2 = new Tiranosaurio();  
  
        System.out.println(anil.toString());  
        System.out.println(anil.morder(ani2));  
        System.out.println(ani1.mover());  
        System.out.println(ani2.toString());  
        System.out.println(ani2.morder(ani1));  
        System.out.println(ani2.mover());  
    }  
}
```

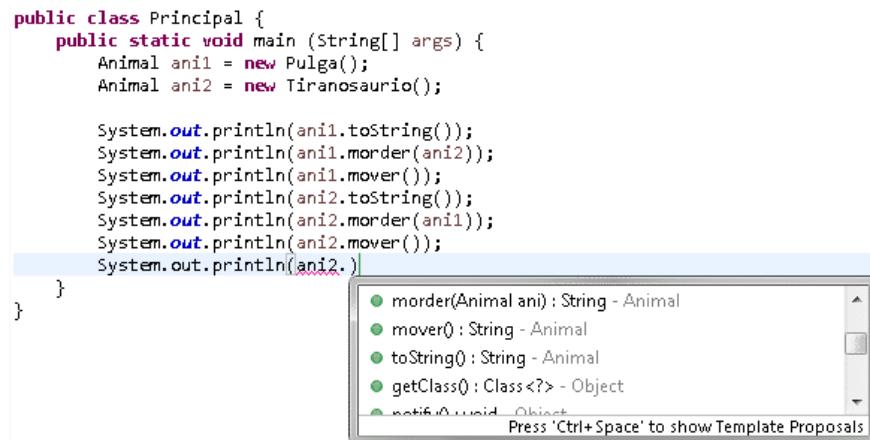
Observa que una referencia de tipo *Animal* puede ser indistintamente una *Pulga* o un *Tiranosaurio*. **Se está aplicado polimorfismo.**

Conversión o cash entre clases

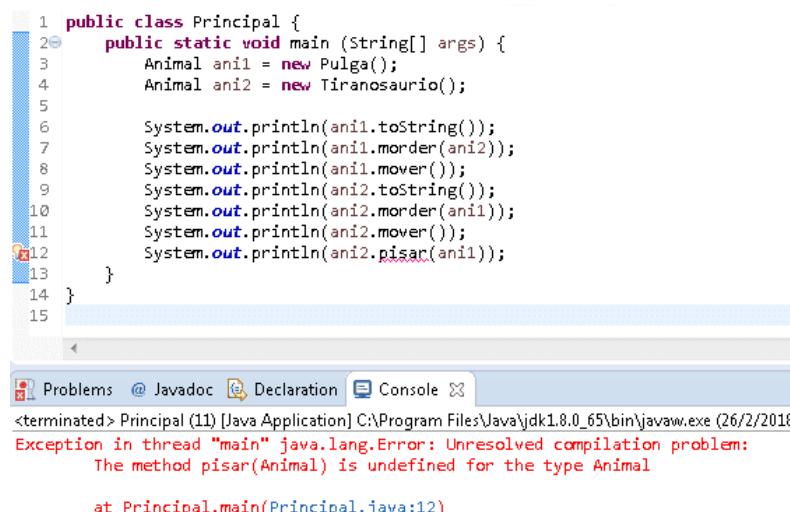
Un objeto *Pulga* y un objeto *Tiranosaurio* tienen ambos la capacidad de *mover* y *morder*, pero además, un *Tiranosaurio* puede *pisar*, y eso es algo que no estaba contemplado en la clase *Animal*. Vamos a ver con qué problema podemos encontrarnos al invocar al método *pisar*.

En primer lugar, ya te habrás dado cuenta de que cuando escribes el nombre de un objeto y un punto, Eclipse te ofrece una lista con los métodos y propiedades para hacerte el trabajo más fácil. Sin embargo, tal como observas en la imagen, no aparece el método *pisar* del objeto *ani2*.

Aunque nosotros sabemos que *ani2* contiene una referencia a un *Tiranosaurio*, no deja de ser de tipo *Animal* y lo que está ofreciendo son las propiedades y métodos de un *Animal*, no de un *Tiranosaurio*.



Aunque escribas la sentencia completamente a mano para invocar al método *pisar*, seguirás teniendo un problema como el que ves en la imagen:



Nosotros sabemos que *ani2* es un *Tiranosaurio*, pero el compilador no lo sabe, y tampoco lo sabe la máquina virtual de Java. Hay que hacer un *cast* o conversión de tipos de la siguiente manera:

(Tiranosaurio) ani2

El código completo de la clase *Principal* quedará así:

```

public class Principal {
    public static void main (String[] args) {
        Animal anil = new Pulga();
        Animal ani2 = new Tiranosaurio();

        System.out.println(anil.toString());
        System.out.println(anil.morder(ani2));
        System.out.println(anil.mover());
        System.out.println(ani2.toString());
        System.out.println(ani2.morder(anil));
        System.out.println(ani2.mover());
        System.out.println((Tiranosaurio) ani2).pisar(anil);
    }
}

```

Puedes construir un objeto Tiranosaurio de dos formas distintas:

A partir de una referencia de tipo *Animal* o a partir de una referencia de tipo *Tiranosaurio*.

```
Animal t1 = new Tiranosaurio();  
Tiranosaurio t2 = new Tiranosaurio();
```

¿Cuándo usar una referencia de tipo más general?

Con una referencia más general nos referimos a una clase abstracta, una interfaz o una clase normal que tiene un grupo de clases derivadas. En nuestro caso, la clase *Animal*.

1. Cuando queremos aplicar polimorfismo, es decir, queremos que nuestra referencia a un objeto *Tiranosaurio* pueda apuntar también a una *Pulga*.

```
Animal ani1 = new Tiranosaurio();  
System.out.println(ani1);  
ani1 = new Pulga();  
System.out.println(ani1);
```

Ten en cuenta que un *Animal* puede ser un *Tiranosaurio*, pero un *Tiranosaurio* no puede ser ni un *Animal* ni una *Pulga*:

```
Tiranosaurio t1 = new Animal();  
Tiranosaurio t2 = new Pulga();
```

2. Cuando no necesitamos los métodos más especializados para cumplir el objetivo del programa. Por ejemplo: no necesitamos que el *Tiranosaurio* tenga la capacidad de *pisar*. Aunque siempre podremos usar el método utilizando la conversión, como has aprendido en este apartado.

¿Cuándo usar una referencia de tipo más específico?

Con una referencia de tipo mas específico nos referimos a las clases derivadas que ocupan los niveles más bajos en la jerarquía. En nuestro ejemplo las clases *Pulga* y *Tiranosaurio*.

1. Cuando no necesitamos aplicar polimorfismo. Quiero una referencia a un objeto *Tiranosaurio* y no necesito que pueda apuntar a un objeto *Pulga*.
2. Quiero utilizar los métodos más específicos, en nuestro ejemplo *pisar*, sin necesidad de tener que hacer una conversión o *cash*.

Las interfaces

Una interfaz, aunque tiene una finalidad muy parecida a una clase abstracta y se utiliza de forma similar, tiene muchas diferencias.

Veamos las características de las interfaces:

1. Una interfaz especifica la forma de sus métodos, pero no da ningún detalle de su implementación; por lo tanto no se puede pensar en ella como la declaración de una clase.
2. Una interfaz no puede tener constructor.
3. Todos los métodos de una interfaz son abstractos, aunque no se incluya la palabra clave *abstract*. Por lo tanto, no se podrá incluir implementación en ninguno de los métodos.
4. Las variables miembro definidas en la interfaz son por defecto finales (constantes, su valor no se puede modificar) aunque se pueden redefinir en las clases derivadas.
5. Una interfaz se define con la palabra clave interfaz, ya que no es una clase.
6. Una interfaz puede heredar de otra interfaz.
7. Una clase abstracta puede implementar una interfaz.

El formato de creación de una interfaz es el siguiente:

```
modificadores interface nombreInterface {  
    // Declaración de variables y métodos  
}
```

Para ponerlo en práctica puedes hacer otro proyecto en Eclipse y crear la interfaz *Vehiculo*. Seleccionando el nombre del proyecto, tendrás que hacer clic derecho y elegir en el menú contextual “*New / Interface*”. Escribe *Vehiculo* como nombre de la interfaz.

```
public interface Vehiculo {  
    int VELOCIDAD_MAXIMA=120;  
  
    String frenar(int cuanto);  
    String acelerar(int cuanto);  
}
```

Una interfaz viene a ser como una declaración de intenciones, algo así como la norma para la creación de un conjunto de clases que deben cumplir unos criterios. Las clases *Moto*, *Coche*, *Camion*, *Avion*, etc., pueden implementar la interfaz *Vehiculo*, lo que significaría que están obligadas a implementar los métodos *acelerar()* y *frenar()* respetando la estructura que dicta la interfaz *Vehiculo*.

Ahora vamos a crear las clases *Coche* y *Moto* implementando la interfaz *Vehiculo*.

Para especificar que una clase implementa una interfaz se utiliza la palabra clave *implements*.

```
public class Coche implements Vehiculo {
    int velocidad = 0;

    public String acelerar(int cuanto) {
        String cadena = "";
        velocidad = velocidad + cuanto;
        if (velocidad > VELOCIDAD_MAXIMA)
            cadena = "Exceso de velocidad ";
        cadena = cadena + "El coche a acelerado y va a " + velocidad
                    + " km/hora";
        return cadena;
    }

    public String frenar(int cuanto) {
        velocidad = velocidad - cuanto;
        return "El coche ha frenado y va a " + velocidad + " km/hora";
    }
}
```

```
public class Moto implements Vehiculo {
    int velocidad = 0;

    public String acelerar(int cuanto) {
        String cadena = "";
        velocidad = velocidad + cuanto;
        if (velocidad > VELOCIDAD_MAXIMA)
            cadena = "Exceso de velocidad ";
        cadena = cadena + "La moto a acelerado y va a " + velocidad
                    + " km/hora";
        return cadena;
    }

    public String frenar(int cuanto) {
        velocidad = velocidad - cuanto;
        return "La moto ha frenado y va a " + velocidad + " km/hora";
    }
}
```

Una clase que implementa una interfaz está obligada a implementar todos los métodos definidos en dicha interfaz.

Ahora podemos declarar un objeto de la clase *Vehiculo* y asignarle indistintamente una *Moto* o un *Coche*, de la misma forma que ocurría con las clases abstractas.

```
public class Principal {
    public static void main (String[] args) {
        Vehiculo v1 = new Moto();
        Vehiculo v2 = new Coche();
        System.out.println(v1.acelerar(100));
        System.out.println(v1.frenar(25));
        System.out.println(v2.acelerar(130));
        System.out.println(v2.frenar(25));
    }
}
```

De nuevo estamos aplicando polimorfismo, ya que una referencia a un objeto *Vehiculo* puede apuntar indistintamente a un *Coche* o a una *Moto*, es decir, puede adaptar múltiples formas.

Las interfaces y la herencia múltiple

Java no admite herencia múltiple, pero sí la implementación de múltiples interfaces y la herencia de una clase base.

Es importante que tengas en cuenta que en el caso de las interfaces no se habla de herencia, sino de implementación. Lo que hacemos es implementar una interfaz que solo nos dicta lo que debe hacer la clase (los métodos que debe tener) pero no cómo hacerlo (la implementación).

Algo así sería correcto en Java:

```
public class Hipopotamo extends Animal implements Mamifero, Anfibio {  
}
```

Las interfaces definen normas o declaraciones de intenciones. Un hipopótamo, aparte de ser un animal, debe comportarse como un mamífero y un anfibio, características que se definen en las interfaces.

Distinguiendo los distintos tipos de polimorfismo

Polimorfismo en tiempo de compilación (sobrecarga)

Recuerda: la sobrecarga es una técnica de POO que permite realizar varias implementaciones para el mismo método.

Veamos un ejemplo para refrescar conceptos:

```
public class Venta {  
    private String producto;  
    private int cantidad;  
    private double precio;  
  
    public Venta(String producto, int cantidad, double precio) {  
        this.producto = producto;  
        this.cantidad = cantidad;  
        this.precio = precio;  
    }  
  
    public Venta(String producto, double precio) {  
        this.producto = producto;  
        this.cantidad = 1;  
        this.precio = precio;  
    }  
  
    public double calcularImporte() {  
        return cantidad * precio;  
    }  
  
    public double calcularImporte(float descuento) {  
        return cantidad * precio - (cantidad * precio * descuento);  
    }  
  
    @Override  
    public String toString() {  
        return "Venta [producto=" + producto + ", cantidad=" + cantidad +  
", precio=" + precio + "]";  
    }  
}
```

Clase con sobrecarga en el constructor y sobrecarga en el método *calcularImporte()*.

La clase *Venta* dispone de dos implementaciones diferentes para el mismo método *calcularImporte*, esto significa que ejecutará una implementación u otra en función de que se

introduzca un argumento o ninguno. También provee dos implementaciones para el método constructor, de modo que podemos construir un objeto *Venta* de dos formas distintas.

En la clase *Principal* podemos construir dos objetos *Venta* de dos formas distintas (indicando la cantidad o sin indicarla, en cuyo caso será 1). También podemos calcular el importe con o sin descuento.

```
public class Principal {
    public static void main(String[] args) {
        Venta pro1 = new Venta("Impresora", 2, 100);
        Venta pro2 = new Venta("Pen Drive", 10);
        System.out.println(pro1.toString());
        System.out.println("Importe Venta: " + pro1.calcularImporte());
        System.out.println(); // Salto de línea
        System.out.println(pro2.toString());
        System.out.println("Importe Venta con descuento: " +
pro2.calcularImporte(0.1f));
    }
}
```

Estamos ante una de las técnicas para lograr polimorfismo, ya que disponemos de métodos que actúan de múltiples formas en función del número y tipo de argumentos que le pasemos.

Ligadura dinámica: referencias polimórficas

La ligadura dinámica es la capacidad de una referencia para apuntar a diferentes tipos de objetos en tiempo de ejecución.

Piensa en el concepto abstracto *Figura*, una figura puede ser un rectángulo, un círculo, un triángulo, etc.

Si implementamos el modelo de clases de imagen, *Figura* puede ser:

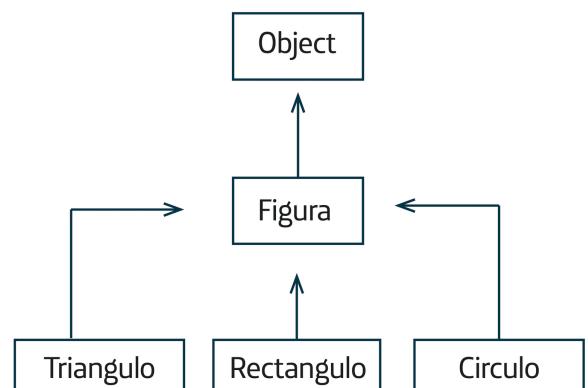
1. Una clase normal, siendo *Triangulo*, *Rectangulo* y *Circulo* sus clases derivadas.

2. Una clase abstracta, siendo *Triangulo*, *Rectangulo* y *Circulo* las clases derivadas que implementan sus métodos abstractos.

Recuerda que no es posible declarar objetos a partir de una clase abstracta, solo a partir de sus derivadas.

3. Una interfaz, siendo *Triangulo*, *Rectangulo* y *Circulo* las clases que implementan la interfaz.

Recuerda que una interfaz solo especifica la forma de sus métodos (cabecera), pero no da ningún detalle de su implementación. La implementación es trabajo para las clases que implementan dicha interfaz.



Ejemplo completo

```
public abstract class Figura {
    private int coordenadaX;
    private int coordenadaY;
    private int ancho;
    private int alto;
    private String tipo;

    public Figura(int coordenadaX, int coordenadaY, int ancho, int alto,
String tipo) {
        this.coordenadaX = coordenadaX;
        this.coordenadaY = coordenadaY;
        this.ancho = ancho;
        this.alto = alto;
        this.tipo = tipo;
    }

    public String consultar() {
        return this.tipo + ": CoordenadaX=" + this.coordenadaX
            + ", CoordenadaY=" + this.coordenadaY + ", Ancho=" +
this.ancho
            + ", Alto=" + this.alto;
    }

    public int getCoordenadaX() {
        return coordenadaX;
    }

    public int getCoordenadaY() {
        return coordenadaY;
    }

    public int getAncho() {
        return ancho;
    }

    public int getAlto() {
        return alto;
    }

    // Método abstracto.
    public abstract double area();
}
```

```
public class Rectangulo extends Figura {

    public Rectangulo(int coordenadaX, int coordenadaY, int ancho, int alto) {
        super(coordenadaX, coordenadaY, ancho, alto, "Rectángulo");
    }

    public double area() {
        return this.getAlto() * this.getAncho();
    }
}
```

```
public class Triangulo extends Figura {
    private int base;
    private int altura;

    public Triangulo(int coordenadaX, int coordenadaY, int ancho, int alto,
int base, int altura) {
        super(coordenadaX, coordenadaY, ancho, alto, "Triángulo");

        this.base = base;
        this.altura = altura;
    }

    @Override
    public String consultar() {
        return super.consultar() + ", Base= " + this.base + ", Altura="
               + this.altura;
    }

    public double area() {
        return this.base * this.altura / 2;
    }
}
```

```
public class Circulo extends Figura {
    private int radio;

    public Circulo(int coordenadaX, int coordenadaY, int ancho, int alto,
                  int radio) {
        super(coordenadaX, coordenadaY, ancho, alto, "Círculo");
        this.radio = radio;
    }

    public double area() {
        return Math.PI * this.radio * this.radio;
    }
}
```

```
public class Principal {

    public static void main(String[] args) {

        Figura fig = new Triangulo(3,3,5,7,5,7);
        System.out.println(fig.consultar());
        System.out.println("Area = " + fig.area());

        fig = new Circulo(7,8,10,10,5);
        System.out.println(fig.consultar());
        System.out.println("Area = " + fig.area());

        fig = new Rectangulo(7,8,7,10);
        System.out.println(fig.consultar());
        System.out.println("Area = " + fig.area());
    }
}
```

Podríamos declarar una referencia a un objeto de tipo *Figura* que dinámicamente podrá cambiar de forma entre *Triangulo*, *Circulo* o *Rectangulo*. Lo puedes comprobar en el siguiente ejemplo:

La variable *fig* es la referencia a nuestro **objeto polimórfico**. En concreto *fig* es una referencia que tiene la capacidad de apuntar a objetos de distinto tipo y, **según el tipo de figura a la que apunta, se ejecutará el método *consultar()* y *area()* que le corresponda, y esto se decide en tiempo de ejecución.**

El resultado del programa será este:

Triángulo: CoordenadaX=3, CoordenadaY=3, Ancho=5, Alto=7, Base= 5, Altura=7

Area = 17.0

Círculo: CoordenadaX=7, CoordenadaY=8, Ancho=10, Alto=10

Area = 78.53981633974483

Rectángulo: CoordenadaX=7, CoordenadaY=8, Ancho=7, Alto=10

Area = 70.0

Una de las grandes ventajas del polimorfismo es la reutilización de código, lo que nos permite escribir el programa anterior de la siguiente manera:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        procesarFigura(new Triangulo(3,3,5,7,5,7));  
  
        procesarFigura(new Circulo(7,8,10,10,5));  
  
        procesarFigura(new Rectangulo(7,8,7,10));  
    }  
  
    static void procesarFigura(Figura f){  
        System.out.println(f.consultar());  
        System.out.println("Area = " + f.area());  
    }  
}
```

Gracias al polimorfismo podemos crear un método común para todos los tipos de figura, dado que las instrucciones a ejecutar son las mismas en los tres casos.

Clases genéricas

Las clases genéricas tienen la posibilidad de declarar una o varias propiedades, cuyo tipo puede variar. El tipo de dicha propiedad será establecido en el momento de crear un objeto.

En el siguiente ejemplo *T* es el tipo genérico, que será reemplazado por un tipo real (*Double*, *Integer*, *String*, *Triangulo*,...).

```
public class Generica<T> {  
    private T dato;  
  
    public Generica(T dato) {  
        this.dato = dato;  
    }  
  
    public String informa() {  
        return "El objeto contiene un dato de tipo: " +  
            this.dato.getClass().getName() +  
            "\nValor = " + this.dato.toString();  
    }  
}
```

La propiedad *dato* es de tipo genérico. Se desconoce el tipo hasta el momento de crear el objeto.

this.dato.getClass().getName() nos suministra el nombre de la clase a la que pertenece el objeto actual (*Double*, *String*, *Integer*, etc.).

Ahora puedes comprobar el funcionamiento de la clase genérica creando la siguiente clase *Principal*:

```
public class Principal {  
    public static void main(String[] args) {  
        Generica<String> miObjeto1 = new Generica<String>("Hola mundo");  
        System.out.println(miObjeto1.informa());  
  
        Generica<Integer> miObjeto2 = new Generica<Integer>(35);  
        System.out.println(miObjeto2.informa());  
  
        Generica<Double> miObjeto3 = new Generica<Double>(45.30);  
        System.out.println(miObjeto3.informa());  
    }  
}
```

Y el resultado obtenido será este:

El objeto contiene un dato de tipo: java.lang.String

Valor = Hola mundo

El objeto contiene un dato de tipo: java.lang.Integer

Valor = 35

El objeto contiene un dato de tipo: java.lang.Double

Valor = 45.3

Comprobación de la clase a la que pertenece un objeto

Comprobación dinámica de tipos

El operador *instanceof* permite comprobar dinámicamente (en tiempo de ejecución) la clase a la que pertenece un objeto con el fin de que el programa pueda responder en consecuencia.

Observa este ejemplo:

```
Figura fig = new Triangulo(3,3,5,7,5,7);
System.out.println(fig.consultar());
System.out.println("Area = " + fig.area());

if (fig instanceof Triangulo) {
    System.out.println("Es un triángulo");
}
```

Introspección y reflexión

En programación orientada a objetos se denomina **introspección** a la capacidad de muchos lenguajes de programación para determinar en tiempo de ejecución la clase a la que pertenece un objeto.

La **reflexión** es la capacidad de inspeccionar y manipular objetos en tiempo de ejecución sin conocer a priori la clase a la que pertenecen dichos objetos.

El ejemplo más sencillo de introspección está en el uso del operador *instanceof* que acabamos de ver, pero también podemos utilizar el método *getClass()* para realizar tareas más sofisticadas. Recuerda que todas las clases heredan de la superclase *Object* y *getClass()* es uno de los métodos heredados de *Object*.

El método *getClass()* devuelve una representación conceptual de la clase a la que pertenece un objeto como un nuevo objeto de tipo *Class*.

En el siguiente ejemplo se comprobará la clase a la que pertenece el objeto apuntado por la referencia *fig* y además se realizará un listado con todos los métodos de la clase aplicando así los principios de la reflexión.

```
import java.lang.reflect.Method;

public class Principal {
    public static void main(String[] args) {
```

```
Figura fig = new Triangulo(3, 3, 5, 7, 5, 7);
System.out.println(fig.consultar());
System.out.println("Area = " + fig.area());

Class clase = fig.getClass();
System.out.println(clase.getName());
Method[] metodos = clase.getMethods();
for (int i = 0; i < metodos.length; i++) {
    System.out.println("Método: " + metodos[i].toString());
}
}
```

El resultado de ejecutar el programa será este:

```
Triángulo: CoordenadaX=3, CoordenadaY=3, Ancho=5, Alto=7, Base= 5, Altura=7
Area = 17.0
Triangulo
Método: public java.lang.String Triangulo.consultar()
Método: public double Triangulo.area()
Método: public int Figura.getCoordenadaX()
Método: public int Figura.getCoordenadaY()
Método: public int Figura.getAncho()
Método: public int Figura.getAlto()
Método: public final void java.lang.Object.wait() throws java.lang.InterruptedException
Método: public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
Método: public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException
Método: public boolean java.lang.Object.equals(java.lang.Object)
Método: public java.lang.String java.lang.Object.toString()
Método: public native int java.lang.Object.hashCode()
Método: public final native java.lang.Class java.lang.Object.getClass()
Método: public final native void java.lang.Object.notify()
Método: public final native void java.lang.Object.notifyAll()
```

Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- La palabra **polimorfismo** es de origen griego y significa **múltiples formas**.
- En el contexto de la programación orientada a objetos el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos y se puede lograr de varias formas:
 - Polimorfismo en tiempo de compilación (**sobrecarga**).
 - Polimorfismo en tiempo de ejecución (**ligadura dinámica**).
 - **Clases genéricas**.
- Para que una referencia tenga la capacidad de ser polimórfica debe ser declarada como:
 - Una clase normal pero que tenga sus clases derivadas.
 - Una clase abstracta.
 - Una interfaz.
- El operador ***instanceof*** permite comprobar dinámicamente (en tiempo de ejecución) la clase a la que pertenece un objeto.
- El método ***getClass()*** devuelve una representación conceptual de la clase a la que pertenece un objeto como un nuevo objeto de tipo *Class*.

5.5. Componentes web con acceso a datos (Servlet, JSP)



Índice

Objetivos	3
Descargar e instalar Apache Tomcat.....	4
Introducción.....	4
Descarga e instalación	5
Servlets.....	6
Introducción.....	6
Configuración del proyecto	6
Nuestro primer Java Servlet.....	15
Generar página de respuesta	18
Ejecución del servlet.....	20
Petición desde un formulario web	24
Ciclo de vida de un servlet.....	28
Acceso a datos desde un servlet	30
Servlet con acceso a base de datos XML.....	31
Java Server Pages (JSP).....	34
Introducción.....	34
Elementos básicos de los documentos JSP	34
Las variables predefinidas request y response	37
Despedida	39
Resumen.....	39

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Desarrollar componentes de servidor (servlets o JSP) que acepten solicitudes de clientes desde sus navegadores web.
- Responder a las solicitudes de los clientes, generando una página HTML dinámica que se enviará como respuesta al servidor.
- Conocer el ciclo de vida de una aplicación web.

Descargar e instalar Apache Tomcat

Introducción

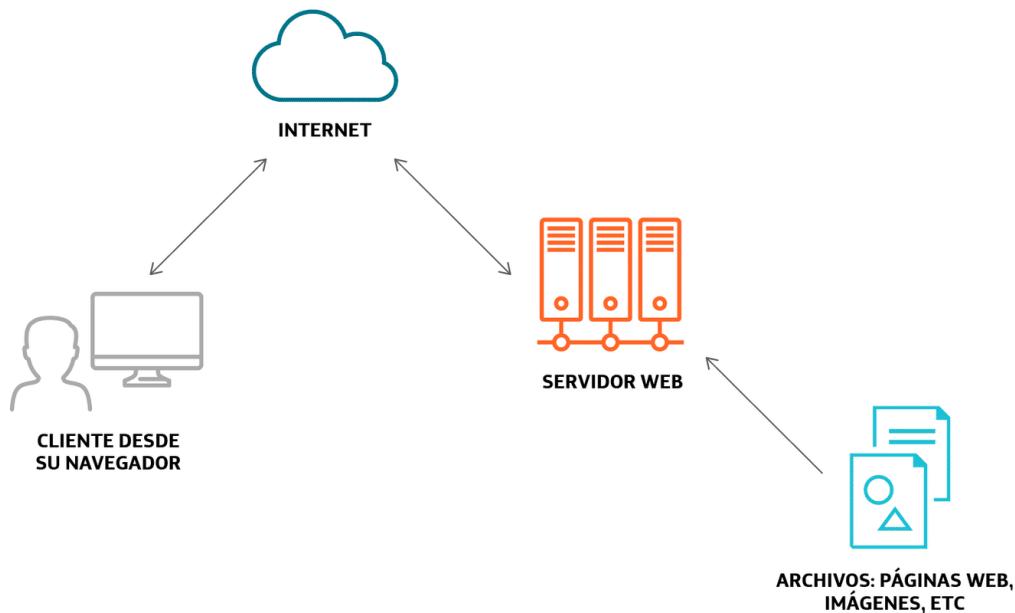
En esta lección, aprenderás a crear aplicaciones que siguen esta lógica:

- El cliente, desde su navegador, hace una petición a través de la URL.
- Un programa de servidor recibe la petición y procesa una respuesta. Aquí es donde debe entrar en juego una de las tecnologías anteriormente mencionadas (servlets, JSP, PHP, etc.). La respuesta finalmente generada tendrá formato HTML.
- El servidor envía la respuesta al cliente, que la visualiza en su navegador.

Pero, ¿para eso necesitamos tener un servidor web en nuestro equipo?

Sí, y por eso vamos a **descargar e instalar el servidor Apache Tomcat**. Así completaremos el ciclo petición-respuesta en nuestro equipo, que actuará como cliente y como servidor.

Recuerda que, para poner a prueba una aplicación web, por pequeña que sea, necesitarás los recursos que ves en la imagen de abajo: un cliente que realice una petición desde su navegador, Internet, un servidor web y el conjunto de recursos que forman el sitio web (páginas, imágenes, programas que forman el *back-end*, etc.).



Estás a punto de descargar e instalar el servidor Apache Tomcat.

¡Adelante!

Descarga e instalación

Puedes descargar Apache Tomcat gratuitamente desde su web oficial.

Accede desde esta URL: <http://tomcat.apache.org/>

Desde ahí, descargaremos la versión 9 ejecutable de Apache Tomcat como servicio web. De este modo se instalará como un servicio más de Windows, y podremos iniciarla y detenerla cuando sea necesario.

Un **servicio Windows** es un **programa que funciona en segundo plano con la finalidad de prestar algún servicio cuando es requerido**. Los servicios pueden iniciarse o detenerse.

La instalación de Apache Tomcat consta de varios pasos, por los que irás avanzando simplemente pulsando el botón "Next". Pero hay dos de ellos a los que merece la pena prestar más atención.

- **Donde se nos solicita la ubicación de la instalación de Java:** Apache Tomcat requiere que esté instalado Java. En la gran mayoría de los casos el instalador detecta la ubicación de la instalación de Java sin mayor problema, pero en algunas ocasiones puede ser necesario especificarla.
- **Donde aparecen los valores de configuración del servidor:** presta atención al valor de "HTTP/1.1 Connector Port", que será 8080. Recuerda que una petición a un servidor se realiza a través del nombre del servidor y un puerto. Pues bien, el nombre del servidor, al tratarse de nuestro propio equipo, será **localhost** y el puerto el **8080**, que es el valor que aparece por defecto.

El siguiente vídeo muestra cómo descargar e instalar Apache Tomcat en tu equipo.

<https://vimeo.com/telefonicaed/review/277073188/38719569a1>

Servlets

Introducción

Los **servlets** son programas que se ejecutan en un servidor web, en una capa intermedia entre una petición proveniente de un navegador web, o cliente, y las bases de datos o aplicaciones del servidor HTTP.

Un servlet puede realizar las **siguientes tareas**:

Leer los datos enviados por el cliente

Que por lo general han sido introducidos en un formulario HTML.

Buscar información respecto a la petición del cliente

Como el *host*, facultades del navegador, las *cookies*, etc.

Generar los resultados

Que se enviarán al cliente, proceso que podría requerir consultar en base de datos o ejecutar otros procesos de la lógica de negocio.

Dar formato a los resultados

Generando dinámicamente un documento, que en la mayoría de los casos tendrá formato HTML.

Establecer los parámetros adecuados para la respuesta HTTP

Esto se traduce en decirle al navegador el tipo de documento que será devuelto (por ejemplo, HTML), entre otras cosas.

Devolver el documento al cliente

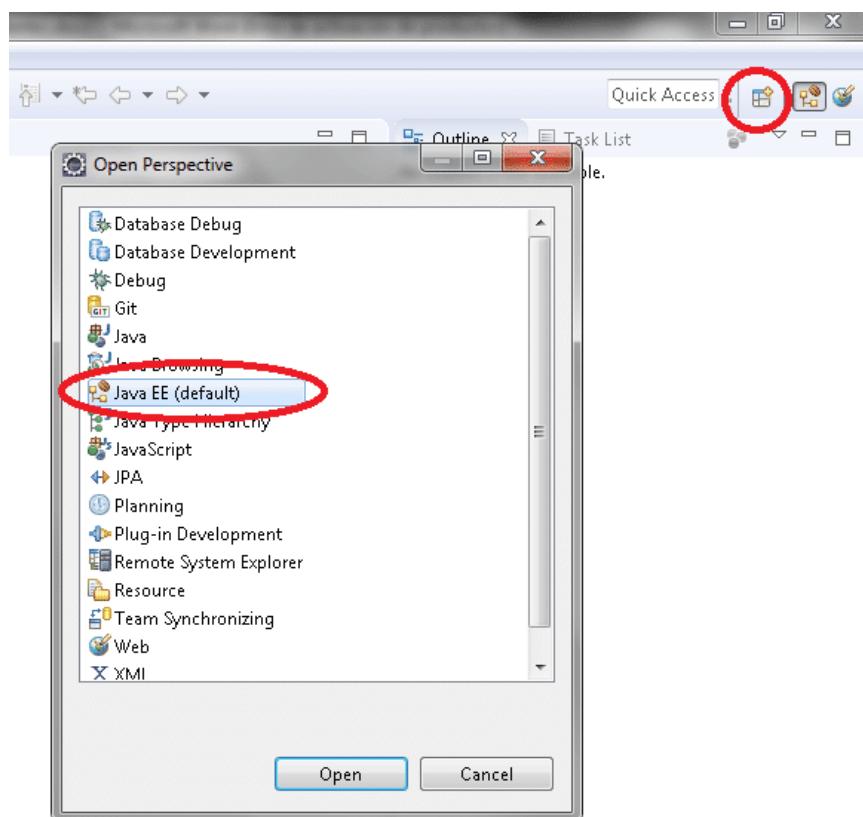
Este documento podría ser enviado en un formato de texto (HTML), binario (imágenes GIF), o incluso en cualquier otro formato comprimido.

Configuración del proyecto

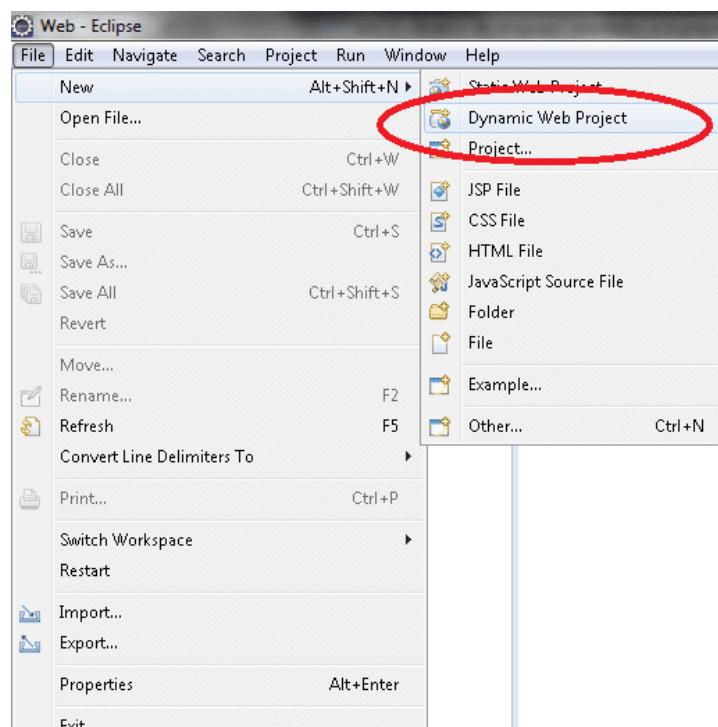
Antes de crear nuestro primer servlet necesitamos un **proyecto Eclipse configurado para que sirva como proyecto web dinámico, capaz de ejecutarse en un servidor web Apache Tomcat**.

Sigue estos pasos:

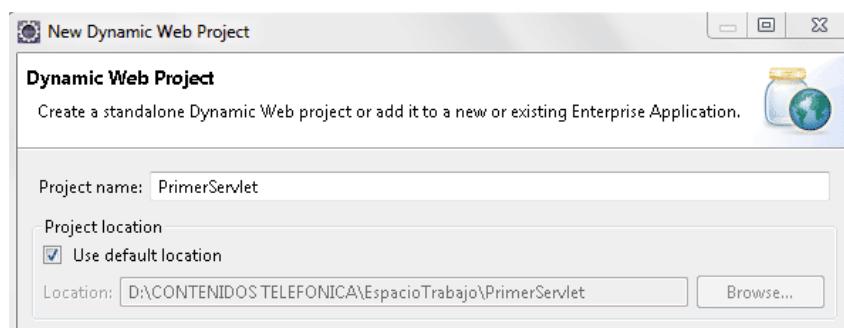
1. Cambia a la perspectiva Java EE, si es que no estás en ella. Es la más adecuada para la tarea que nos ocupa. Pulsa el botón **Open Perspective** y selecciona **Java EE** en el cuadro de diálogo que te aparecerá.



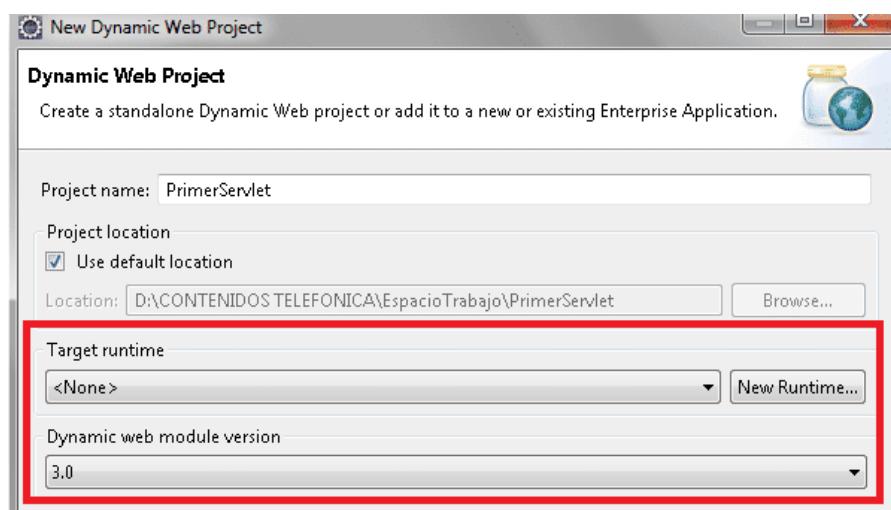
2. Selecciona en la barra de menú **File / New / Dynamic Web Project**.



3. En el cuadro de diálogo *Dynamic Web Project* debes comenzar por **escribir el nombre del nuevo proyecto**. Observa que el cuadro *Location* te está mostrando la ubicación en la que se creará el proyecto, que será a partir de la ubicación del espacio de trabajo de Eclipse.

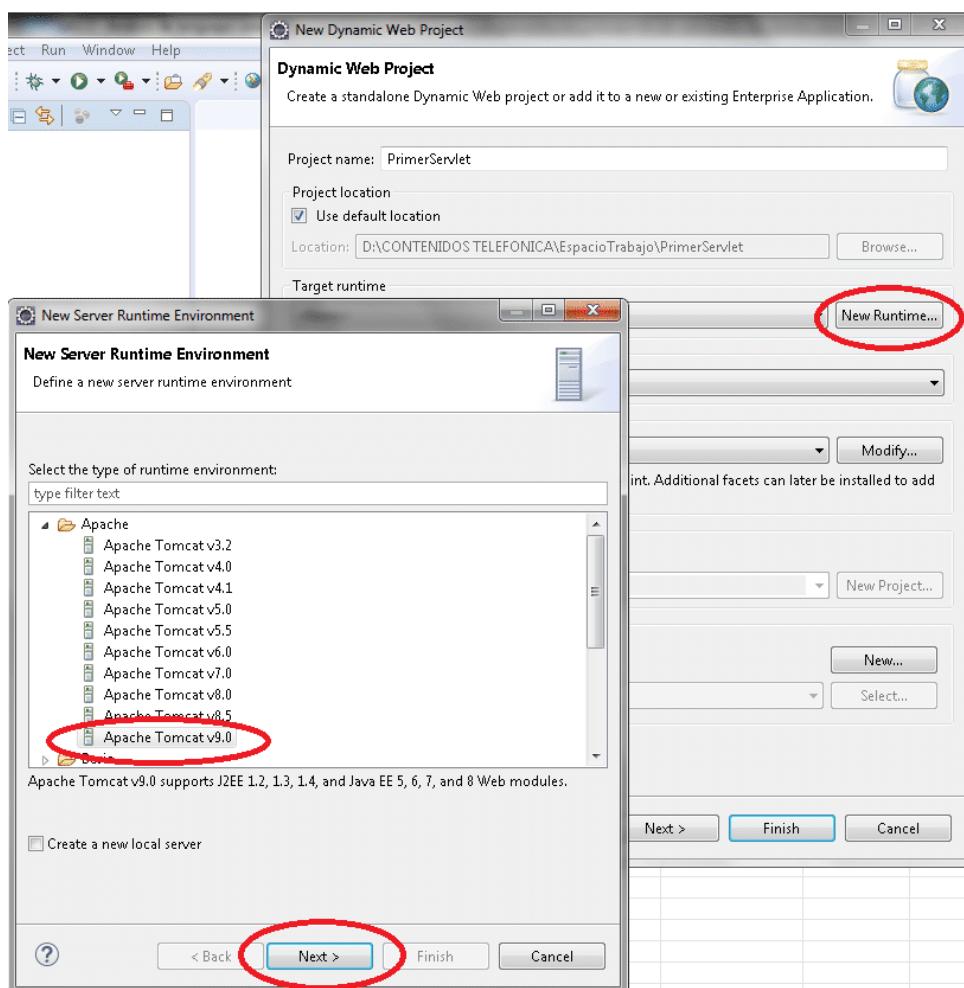


4. Ahora, vamos a prestar atención a las configuraciones del **Target Runtime** y **Web Module Version**.



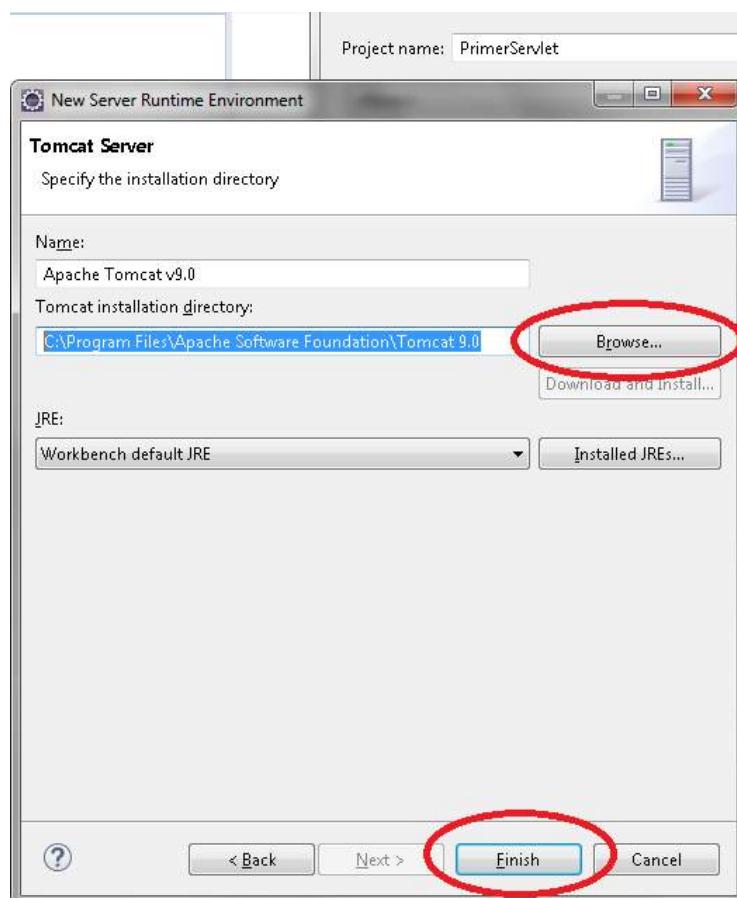
- El **target runtime** hace referencia al responsable de la ejecución del código Java que generaremos para este proyecto. Puesto que en esta ocasión vamos a generar código de servidor o *back-end*, el responsable de la ejecución será el servidor, es decir, Apache Tomcat.
- El **Dynamic web module versión** hace referencia a cómo se comportará Eclipse a la hora de generar código automático. Nosotros no tendremos que escribir todo el código de nuestros servlets, sino que gran parte del código será generado automáticamente por Eclipse, y dicho código variará ligeramente en función del número de versión.

Al tratarse de nuestro primer proyecto web dinámico, no tenemos nada en la lista desplegable **Target runtime**. **Pulsa el botón New Runtime** para obtener el cuadro de diálogo **New Server Runtime Environment**.



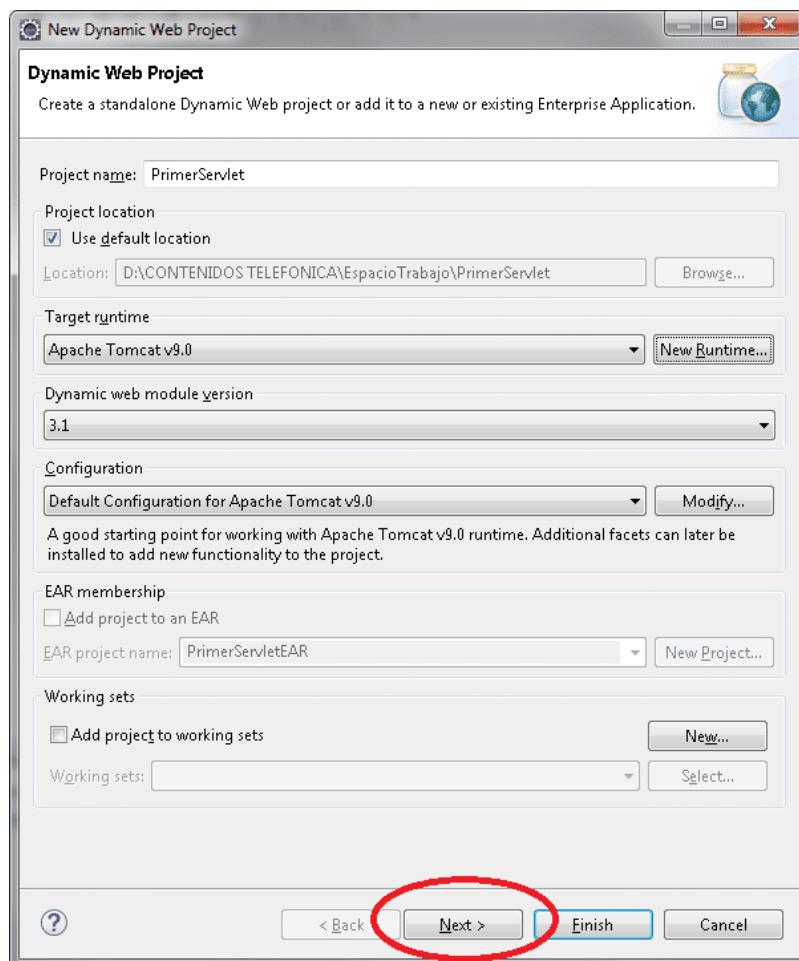
5. Selecciona **Apache Tomcat v9.0**. Si trabajas con una versión antigua de Eclipse, podría ser que no reconozca que existe la versión 9. En ese caso, no te quedaría más remedio que descargar y descomprimir una versión de Eclipse más moderna. Una vez seleccionada la versión 9 de Apache Tomcat haz clic en el botón **Next**.

6. Ahora, Eclipse te está pidiendo que especifiques la ubicación de la instalación de Apache Tomcat y debes pulsar el botón **Browse...** para buscar dicha ubicación. Si utilizas Windows, lo más habitual es que se encuentre en “C:\Program Files\Apache Software Foundation\Tomcat 9.0”.



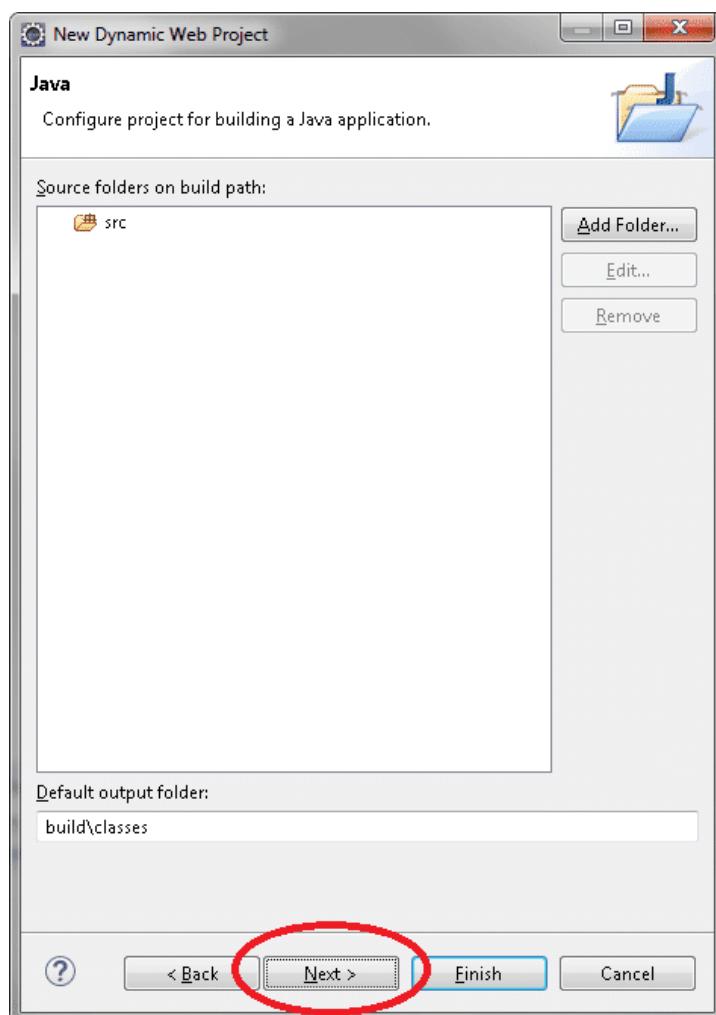
Una vez especificada la ubicación, pulsa el botón **Finish**.

7. Ahora, el cuadro de diálogo *New Dynamic Web Project* muestra “Apache Tomcat v9.0” como *Target Runtime*.



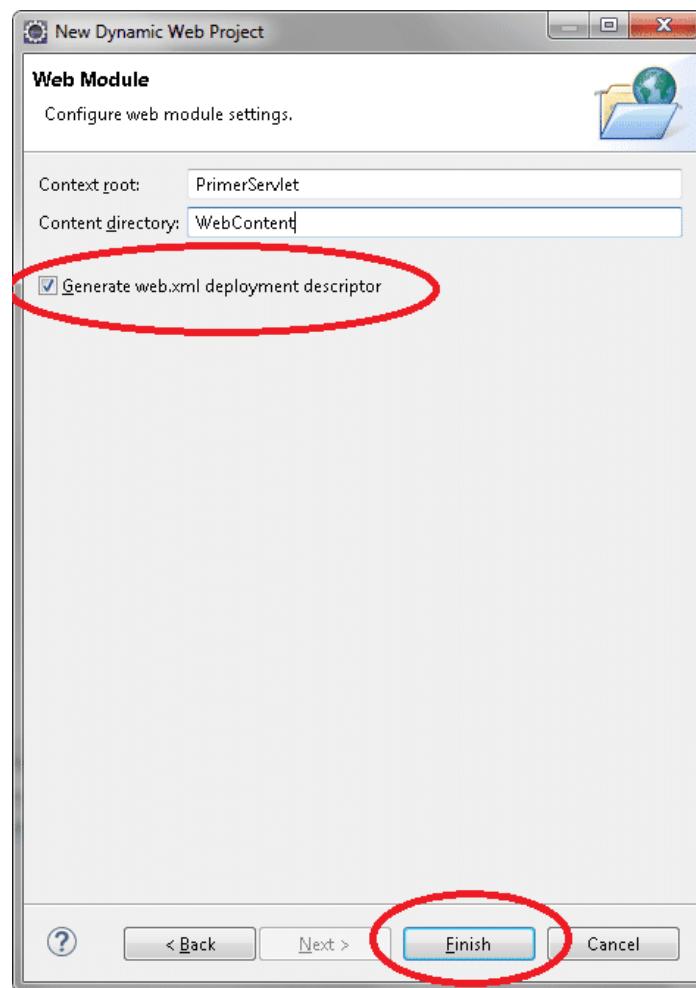
Pulsa el botón **Next** para ir al siguiente paso.

- 8.** En esta ocasión, Eclipse simplemente te está informando sobre la ubicación del código fuente de los programas Java, que es en la carpeta **src**; no tienes que hacer nada especial.



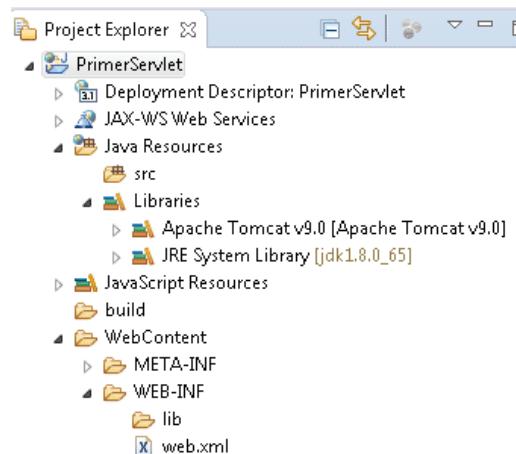
Sólo pulsa el botón **Next** para avanzar al siguiente paso.

9. Ahora, es importante que actives la casilla de verificación ***Generate web.xml deployment descriptor***. De esta forma, Eclipse generará por nosotros el archivo web.xml, que contendrá valores de configuración del sitio web.

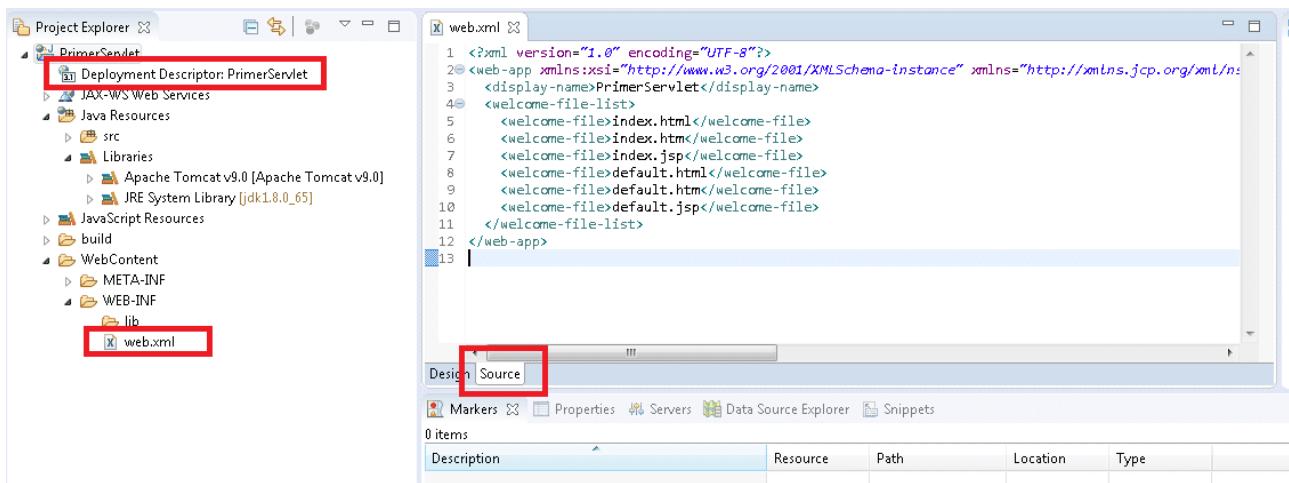


Por último, haz clic en el botón **Finish** y tendrás creado un proyecto Java, configurado para crear una aplicación web dinámica que se ejecutará en Apache Tomcat.

10. Fíjate ahora en el explorador de proyectos para familiarizarte con la estructura que tiene un proyecto web dinámico. Expande las carpetas que necesites hasta que el proyecto aparezca como en la siguiente imagen:



- La carpeta **Java Resources**, como su nombre indica, contiene todos los recursos Java. Está compuesta por las subcarpetas **src**, donde se guardarán tus códigos fuentes, y **Libraries**, donde se encuentran las librerías necesarias. En este caso, además de las librerías del JRE, se encuentran otras librerías suministradas por el servidor Apache Tomcat.
- La carpeta **Web Content** es donde se irán ubicando todos los archivos del sitio web, que pueden ser ficheros HTML, imágenes, archivos CSS, etc. como sabes por lecciones anteriores.
- Observa que dentro de *Web Content* hay una carpeta llamada **WEB-INF** que contiene el archivo **web.xml**. Como hemos mencionado en otras ocasiones, este archivo contiene configuración del sitio web que será relevante para el servidor. Ábrelo para ver su contenido, pero tendrás que situarte en la pestaña *Source* para verlo igual que en la siguiente imagen:



Otra forma de abrir el archivo web.xml es haciendo clic en el enlace *Deployment Descriptor: PrimerServlet*. Por ahora solamente vamos a analizar esta línea:

<welcome-file>index.html</welcome-file>

Estamos indicando el nombre de la página de inicio, o *home page*, de nuestra web. El archivo web.xml, generado automáticamente, ofrece varias alternativas de nombres. El servidor HTTP actúa de la siguiente manera: si un cliente accede desde su navegador a la dirección www.unaurl.com, está indicando el nombre de dominio, pero no el recurso al que desea acceder. En este caso, si el proyecto alojado en dicho dominio tiene un archivo web.xml, como el de la anterior imagen, devolvería al cliente el contenido del archivo index.html en caso de que exista, y, si no existiese, devolvería el contenido del archivo index.htm. Y así sucesivamente.

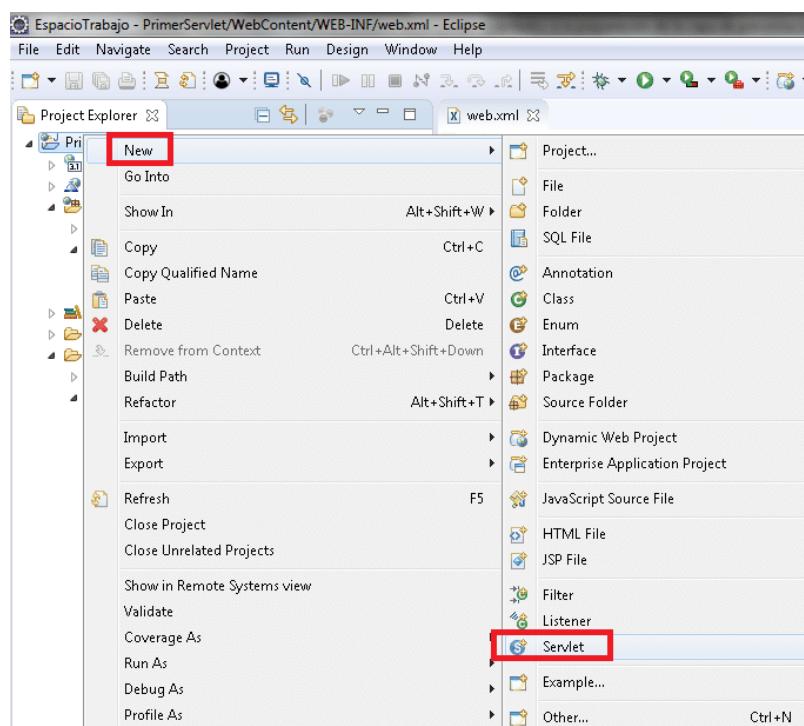
Ya tienes el proyecto perfectamente configurado. Sólo falta crear el *servlet* para ponerlo a funcionar.

Nuestro primer Java Servlet

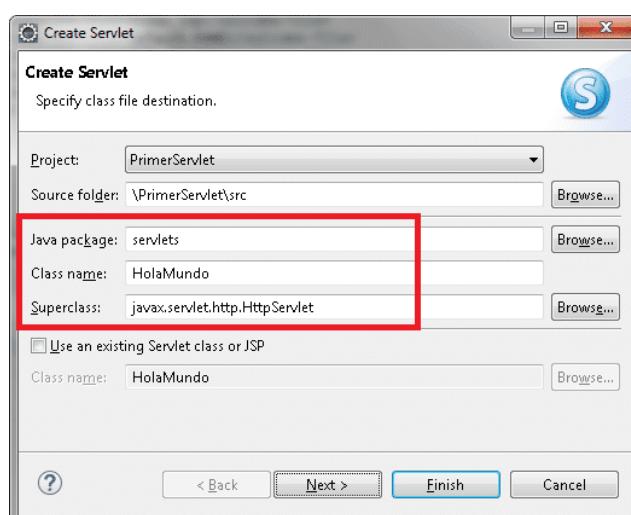
Ha llegado el momento de crear el primer *servlet* que nos sirva para atender peticiones desde un navegador web. Lo llamaremos **HolaMundo**.

Sigue estos pasos:

1. Con el puntero del ratón sobre el nombre del proyecto, haz **clic derecho** y selecciona **New / Servlet** en el menú contextual.



2. Debes tener abierto el cuadro de diálogo **Create Servlet**, en el que necesitarás establecer algunos valores. Observa la imagen:



Los valores de configuración del *servlet* que te interesan en este momento son los siguientes:

Java package

Paquete Java donde se ubicará el *servlet*. Al fin y al cabo, un *servlet* es una clase Java que podrá estar ubicada en un paquete. **Escribe *servlets* como nombre de paquete.**

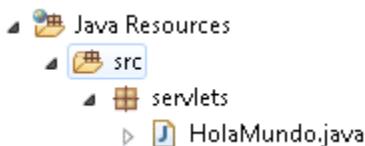
Class name

Será el nombre de la clase que representará al *servlet*. Escribe aquí **HolaMundo**.

Superclass

Aunque aquí no tienes que tocar nada, es interesante que prestes atención. Un *servlet* es una clase Java que deriva de la superclase **HttpServlet**, que está ubicada en el paquete o librería **javax.servlet.http**.

3. Pulsa *Finish* para terminar y despliega la carpeta Java Resources de tu proyecto. Gran parte de tu *servlet* ya estará creado. En la carpeta *src* se ha creado la estructura, conforme a la configuración que especificaste: una clase Java llamada **HolaMundo en un paquete llamado **servlets**.**



Tu nuevo *servlet* tendrá el siguiente aspecto:

```

package servlets;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class HolaMundo
 */
@WebServlet("/HolaMundo")
public class HolaMundo extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public HolaMundo() {
        super();
        // TODO Auto-generated constructor stub
    }
}
    
```

```
 /**
 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
response)
 */
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    // TODO Auto-generated method stub
    response.getWriter().append("Served at:
").append(request.getContextPath());
}

/**
 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
response)
 */
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    // TODO Auto-generated method stub
    doGet(request, response);
}

}
```

A continuación, vamos a analizar el código que ha generado Eclipse de manera automática.

```
package servlets;
```

La nueva clase Java estará ubicada en el paquete *servlets*, tal como indicamos en el momento de su creación.

```
import java.io.IOException;
```

Ya que el objetivo de cualquier *servlet* es atender peticiones de clientes y responder a dichas peticiones con una página de respuesta, es fácil predecir que, en estas tareas, entran en juego operaciones de entrada y salida de datos para la comunicación entre cliente y servidor. Estas operaciones de entrada/salida podrían ocasionar excepciones que podrían ser capturadas por un objeto de la clase *IOException*.

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

Todas las clases que forman parte de la jerarquía de clases relacionadas con los *servlets* se encuentran en la librería *javax.servlet*. Poco a poco irás aprendiendo la función de todas estas clases.

```
@WebServlet("/HolaMundo")
public class HolaMundo extends HttpServlet { }
```

Lo que hace que nuestra clase pueda considerarse un *servlet* es que extiende la **superclase *HttpServlet***, lo que la provee de la funcionalidad necesaria para poder considerarse un *servlet*.

La **anotación *@WebServlet*** se utiliza para registrar el *servlet* en el servidor. En el *argumento*, especificamos el nombre que utilizará el cliente para hacer referencia al *servlet* desde el navegador. Para acceder al *servlet* desde el navegador, utilizaríamos el siguiente formato:

protocolo://servidor:puerto/proyecto/Servlet

Que para nuestro ejemplo sería:

http://localhost:8080/PrimerServlet/HolaMundo

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    doGet(request, response);
}
```

Recuerda que la **misión principal de un *servlet* es atender la petición de un cliente desde su navegador web** a través del protocolo HTTP y esta petición suele llegar en la mayoría de los casos desde la URL.

Pues bien, **cada *servlet* está siempre en ejecución en el servidor a la espera de atender peticiones**. Cada vez que recibe una petición, ésta es atendida en los métodos ***doGet()*** o ***doPost()***. El método *doGet()* atiende peticiones de tipo GET, y *doPost()* atiende peticiones de tipo POST.

Más adelante aprenderás la diferencia entre una petición GET y una petición POST. Pero en nuestro ejemplo trataremos de la misma forma una petición GET que una petición POST, ya que, si te fijas, desde el método *doPost()* invocamos al método *doGet()*, donde falta completar el código para generar la página de respuesta que se enviará al cliente.

Generar página de respuesta

En este apartado **completaremos el código del método *doGet()* de nuestro *servlet*** para que responda a la petición del cliente.

Comienza por sustituir el método *doGet()* actual por el siguiente:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter flujoEscritura=response.getWriter();

    flujoEscritura.append("<!DOCTYPE html>");
    flujoEscritura.append("<html><head><meta charset='UTF-8'>");
    flujoEscritura.append("<title>Página dinámica</title>");
    flujoEscritura.append("</head><body>");
    flujoEscritura.append("<h1>Hola Mundo</h1>");
    flujoEscritura.append("</body></html>");
    flujoEscritura.close();

}
```

Ahora, vamos a analizar detenidamente la implementación del método:

1.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
```

En primer lugar, el método *doGet()* recibe dos argumentos: ***request*** de tipo *HttpServletRequest* y ***response*** de tipo *HttpServletResponse*.

El objeto ***request* hace referencia a la petición realizada por el usuario**. A través de este objeto podemos obtener información sobre el cliente que realizó la solicitud: navegador, parámetros de entrada, etc.

El objeto ***response* hace referencia a la respuesta que enviará el servidor** y nos servirá para preparar la página HTML de respuesta.

```
response.setContentType("text/html");
```

Con esta línea estamos indicando el formato de la página de respuesta que vamos a enviar al cliente. El texto especificado en el argumento es lo que se denomina un tipo [mime](#).

```
PrintWriter flujoEscritura=response.getWriter();
```

Enviar la página de resultado al cliente implica una operación de entrada y salida de datos, y eso requiere del uso de un **flujo o stream**. Toda información que se transmite entre ordenadores fluye desde una entrada hacia una salida, y esta trasmisión ha de ser gestionada por un objeto especial que represente dicho flujo o *stream*. Será el objeto *response* quien nos provea de dicho flujo, ejecutando el método ***getWriter()***, que retornará un objeto ***PrintWriter***.

```
flujoEscritura.append("<!DOCTYPE html>");  
flujoEscritura.append("<html><head><meta charset='UTF-8'>");  
flujoEscritura.append("<title>Página dinámica</title>");  
flujoEscritura.append("</head><body>");  
flujoEscritura.append("<h1>Hola Mundo</h1>");  
flujoEscritura.append("</body></html>");
```

A partir de aquí, la tarea consiste en utilizar el método **append(...)** para ir enviando cadenas de texto que formarán un documento HTML completo.

```
flujoEscritura.close();
```

Para finalizar, cerramos el flujo de datos.

**Sigue adelante para ejecutar tu primer *servlet*.
¡Ya queda muy poco!**

Ejecución del servlet

En este apartado podrás ver el resultado final de la ejecución del *servlet*.

Recuerda que nuestro *servlet* será ejecutado en el servidor, es decir, será responsabilidad de Apache Tomcat ejecutarlo.

En primer lugar, debes tener en cuenta que Apache Tomcat actúa como servicio y, en consecuencia, dicho servicio puede iniciarse o detenerse. Pues bien, Eclipse se encargará de iniciar y detener el servicio cuando sea necesario, pero requiere que no esté iniciado ya desde fuera. Si trabajas con Windows, puedes comprobar si Apache Tomcat está iniciado abriendo la herramienta *Monitor Tomcat*.

Sigue los pasos que te indicamos para ejecutar el *servlet*. Lo primero, será detener Apache Tomcat.

1. Detener Apache Tomcat

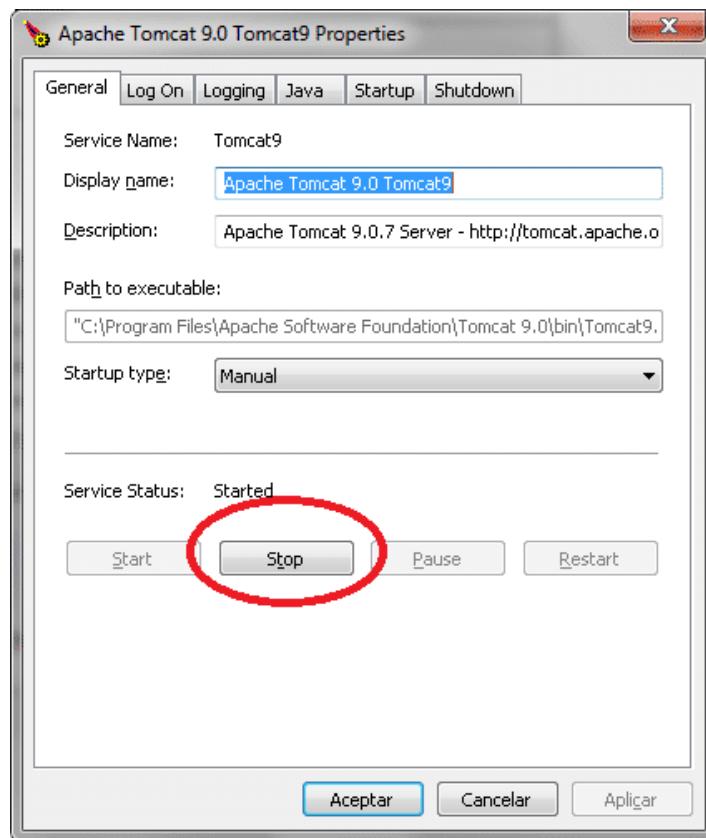
Utiliza el cuadro de texto de **Buscar programas y archivos**.



Escribe **Monitor Tomcat** y pulsa enter.

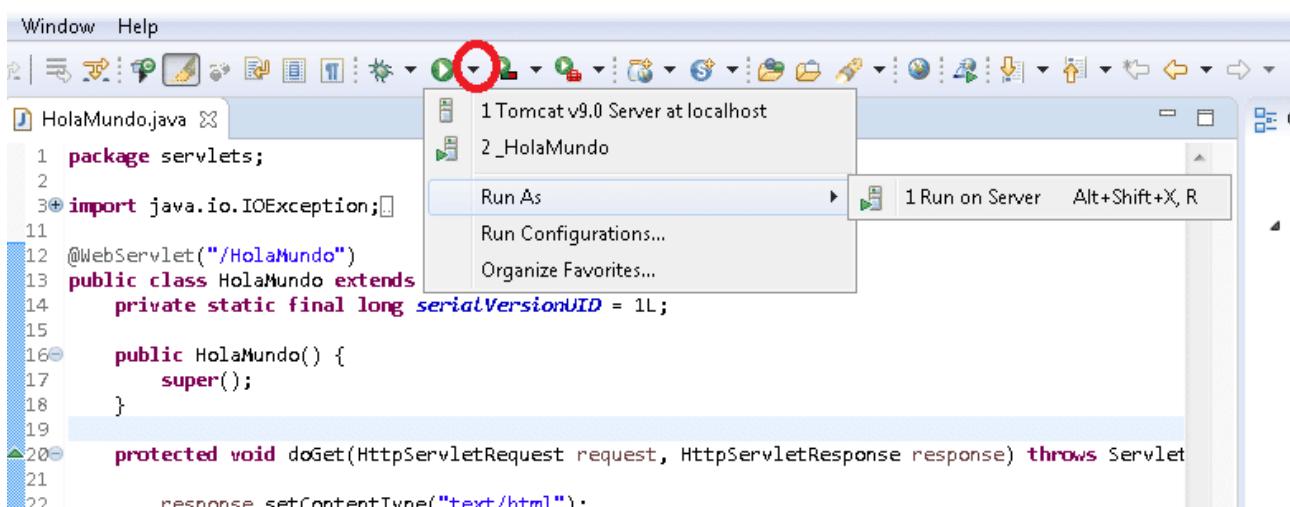


Ahora, pulsa el botón **Stop** del cuadro de diálogo *Apache Tomcat 9.0 Properties*.

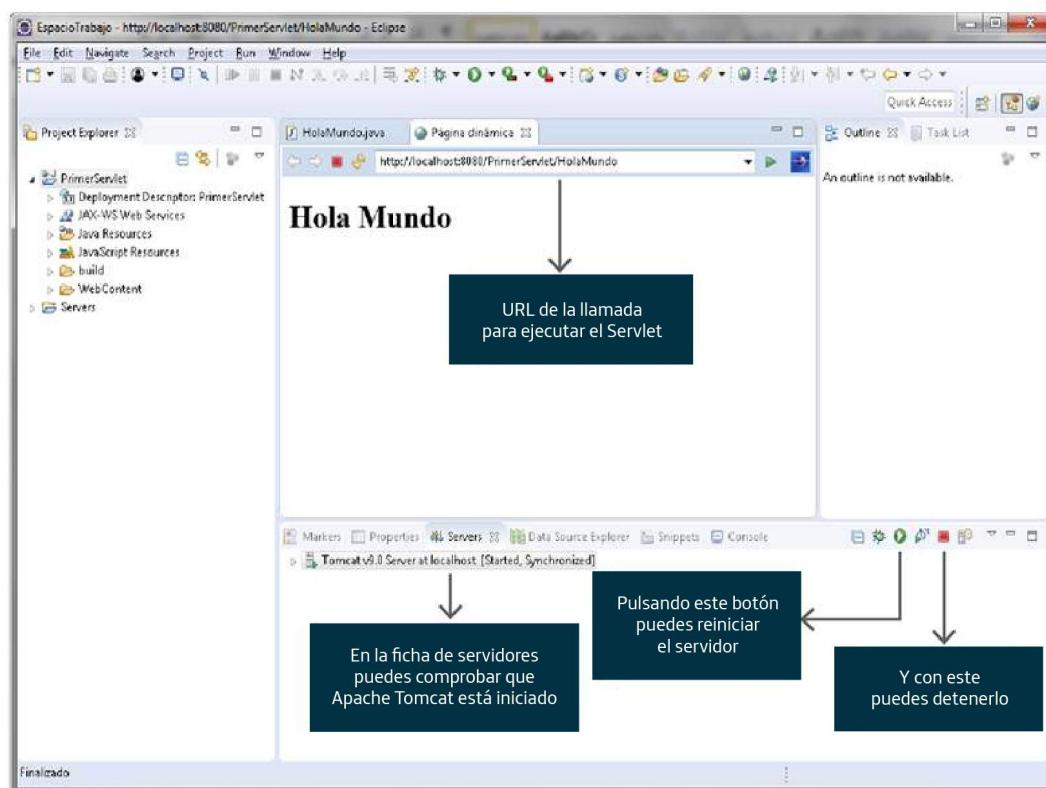


2. Ejecutar el proyecto

Haz clic en el icono del triángulo negro, en el botón de ejecutar, y selecciona **Run As / Run on Server** en el menú contextual.

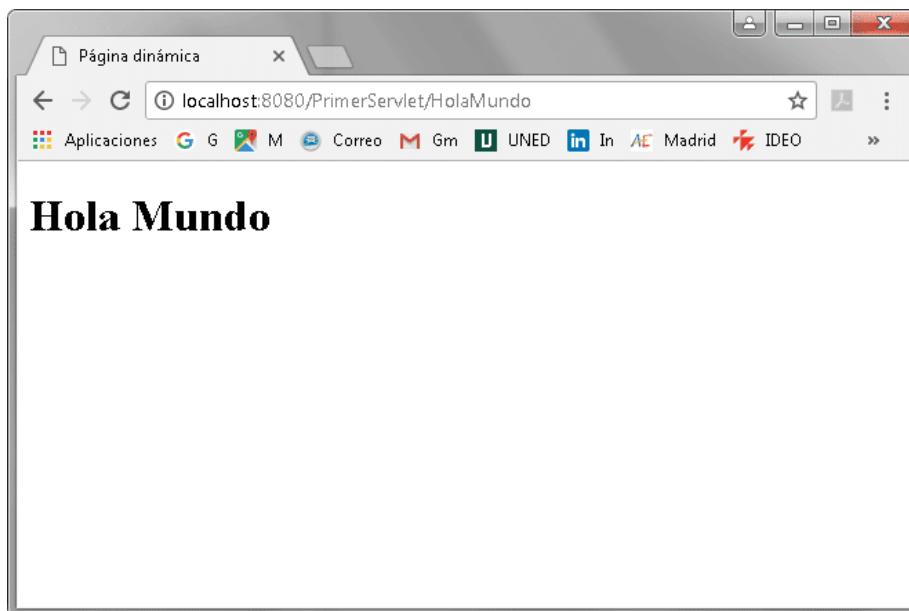


Si has seguido bien los pasos, Eclipse te estará mostrando el resultado en el explorador que tiene integrado.



Desde Eclipse puedes interactuar con el servidor Apache mediante la ficha **Servers**: puedes comprobar que está iniciado, puedes detenerlo, reiniciarlo, etc.

El **servlet** ha quedado registrado en el servidor y está preparado para recibir solicitudes de múltiples clientes desde sus navegadores. Incluso puedes probar a escribir la URL desde Google Chrome o cualquier otro navegador.



Ejecución desde Google Chrome.

Si ahora quisieras modificar tu *servlet* para que, por ejemplo, escriba varias veces “Hola Mundo” en varios tamaños, sería necesario detener el servidor y volver a iniciar lo para que el nuevo *servlet* con las modificaciones quede registrado correctamente.

Modifica de nuevo el método *doGet()* de la siguiente manera:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter flujoEscritura=response.getWriter();

    flujoEscritura.append("<!DOCTYPE html>");
    flujoEscritura.append("<html><head><meta charset='UTF-8'>"); 
    flujoEscritura.append("<title>Página dinámica</title>"); 
    flujoEscritura.append("</head><body>"); 
    for (int i=1; i<7; i++) {
        flujoEscritura.append("<h"+i+">Hola Mundo</h"+i+">"); 
    }
    flujoEscritura.append("</body></html>"); 
    flujoEscritura.close();
}
```

Una vez que hayas terminado los cambios, haz clic en el botón **Stop Server**.



Vuelve a ejecutar el *servlet* de la misma forma que lo hiciste anteriormente y el resultado será:

Hola Mundo

Hola Mundo

Hola Mundo

Hola Mundo

Petición desde un formulario web

Vamos a completar el proyecto anterior creando el archivo *index.html*, que incluirá un enlace al *servlet* anterior y un formulario que realizará la petición a un segundo *servlet*.

El archivo *index.html* incluirá los siguientes elementos:

- Un vínculo, que realizará la petición al *servlet HolaMundo*.
- Un formulario, que permitirá al usuario introducir los tres lados de un triángulo. Al enviar dicho formulario se ejecutará un *servlet*, que recibirá los tres parámetros y generará una página de respuesta.

Recuerda que debes hacer clic derecho sobre el nombre del proyecto y seleccionar **New / HTML file** en el menú contextual. Luego, puedes simplemente copiar y pegar este código:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Triángulos</title>
</head>
<body>
<h1>Triángulos</h1>
<p>
    <a href="HolaMundo">Ejecuta el Servlet HolaMundo</a>
</p>
<form action="Triang" method="get">
<fieldset>
<legend>Introduce los tres lados del triángulo</legend>
<p>
    <label for="lado1">Lado 1:</label>
    <input type="number" name="lado1" />
</p>
<p>
    <label for="lado2">Lado 2:</label>
    <input type="number" name="lado2" />
</p>
<p>
    <label for="lado3">Lado 3:</label>
    <input type="number" name="lado3" />
</p>
<p>
    <input type="submit" value="Enviar" />
    <input type="reset" value="Limpiar" />
</p>
</fieldset>
</form>
</body>
</html>
```

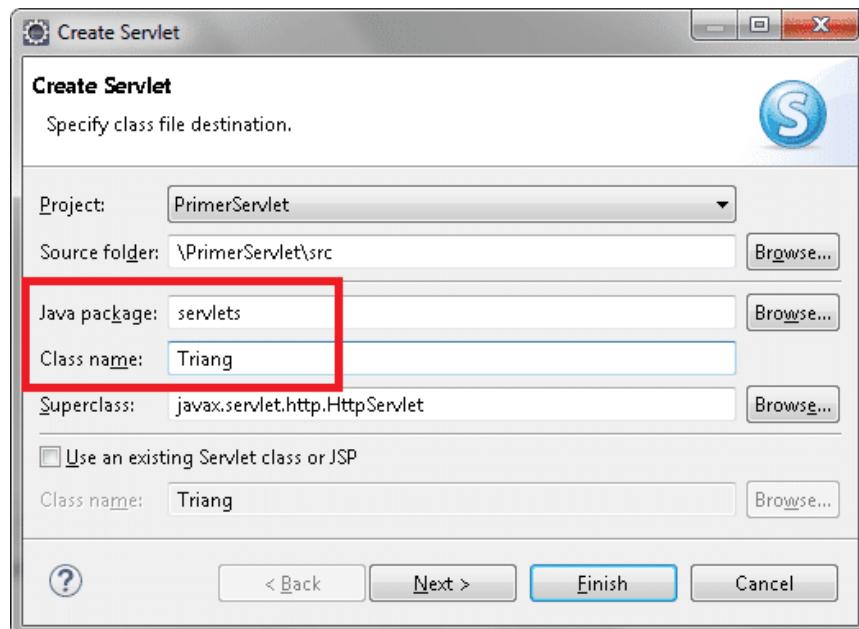
Tu nuevo documento HTML se visualizará de la siguiente manera:

The screenshot shows a web browser window with three tabs: 'index.html', 'HolaMundo.java', and 'Triángulos'. The 'Triángulos' tab is active, displaying the URL 'http://localhost:8080/PrimerServlet/index.html'. The page content includes the title 'Triángulos' and a link 'Ejecuta el Servlet HolaMundo'. Below this is a form with the instruction 'Introduce los tres lados del triángulo'. It contains three input fields labeled 'Lado 1:', 'Lado 2:', and 'Lado 3:'. At the bottom are two buttons: 'Enviar' (Send) and 'Limpiar' (Clear).

El vínculo ya está preparado para ejecutar el anterior *servlet*.

¡Ahora te toca crear el segundo!

Crea tu *servlet* igual que lo hiciste la vez anterior. Recuerda crearlo dentro del paquete *servlets* y llamarlo **Triang**, tal como nos hemos referido a él en el documento HTML.



Puedes copiar y pegar el código desde aquí:

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Triang")
public class Triang extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Triang() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

        int lado1, lado2, lado3;

        response.setContentType("text/html");
        PrintWriter flujoEscritura=response.getWriter();

        flujoEscritura.append("<!DOCTYPE html>");
        flujoEscritura.append("<html><head><meta charset='UTF-8'>");
        flujoEscritura.append("<title>Página dinámica</title>");
        flujoEscritura.append("</head><body>");

        try {
            lado1 = Integer.parseInt(request.getParameter("lado1"));
            lado2 = Integer.parseInt(request.getParameter("lado2"));
            lado3 = Integer.parseInt(request.getParameter("lado3"));
            flujoEscritura.append("<h2>El triangulo con lados " + lado1 + ", " +
lado2 + ", " + lado3 + " es ");
            if (lado1==lado2 && lado2==lado3)
                flujoEscritura.append("EQUILÁTERO</h2>");
            else if (lado1==lado2 || lado2==lado3 || lado1==lado3)
                flujoEscritura.append("ISÓSCELES</h2>");
            else
                flujoEscritura.append("ESCALENO</h2>");

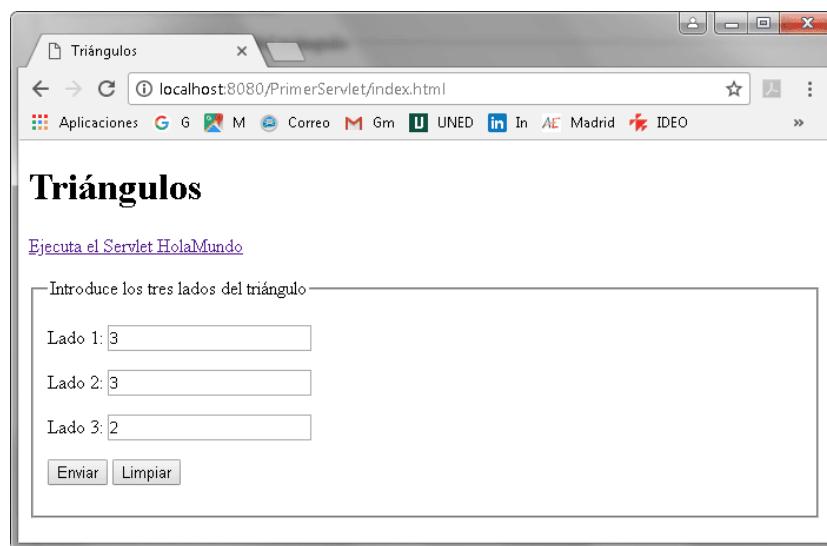
        } catch (NumberFormatException e) {
            flujoEscritura.append("Alguno de los lados que has introducido no es
correcto");
        }

        flujoEscritura.append("</body></html>");
        flujoEscritura.close();
    }

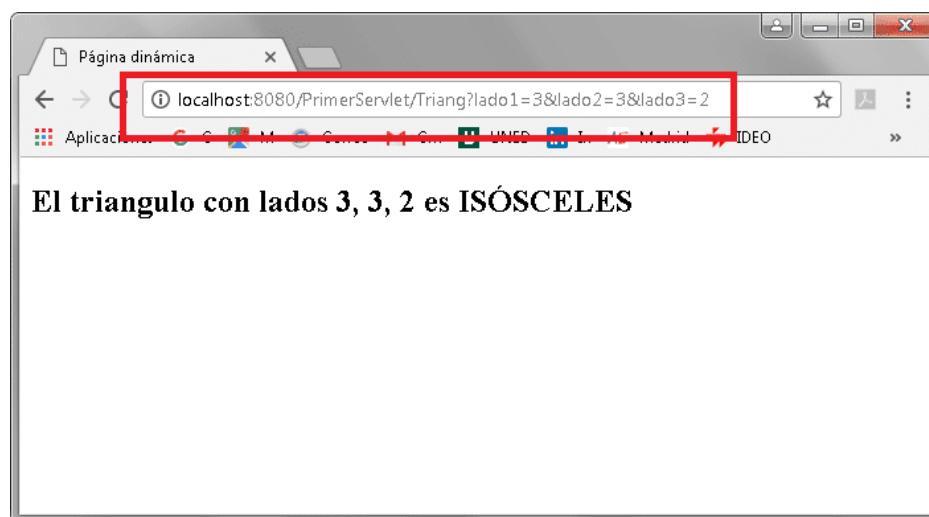
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Ahora, es importante que prestes atención a lo que está ocurriendo cuando el usuario hace clic en el botón *Enviar* del formulario. Puede resumirse en estos puntos:

1. El usuario escribe la longitud de los tres lados:



2. Después, el usuario hace clic en el botón **Enviar**, lo que provoca la **redirección** al **servlet Triang**.



En la nueva URL, no sólo aparece el nombre del servlet, sino que además se pasan como parámetros los valores que ha introducido el usuario y el servlet es capaz de recoger dichos parámetros para responder al usuario. Los parámetros son recogidos dentro del servlet por medio del objeto **request** y el método **getParameter()**.

```
lado1 = Integer.parseInt(request.getParameter("lado1"));
lado2 = Integer.parseInt(request.getParameter("lado2"));
lado3 = Integer.parseInt(request.getParameter("lado3"));
```

Vamos a repasar el sistema que utilizamos desde el formulario HTML para provocar la ejecución del servlet.

```
<form action="Triang" method="get">
```

En el parámetro **action** indicaste el nombre del *servlet*, y en el parámetro **method**, el método de envío. El método *get* hace que los parámetros que envía el usuario se pasen a través de la URL. Otra posibilidad es usar el método *post*, que hace que los parámetros se envíen en el cuerpo de la petición y, por lo tanto, no se muestren en la URL. **Este sistema sería más conveniente para datos que no nos interesa que se vean por temas de seguridad y privacidad.**

Ciclo de vida de un servlet

Cuando no estás familiarizado con el desarrollo de aplicaciones web, puede resultar complejo comprender su ciclo de vida.

Puede que estés acostumbrado a que un programa comienza, realiza las tareas que tiene encomendadas y termina. El **ciclo de vida de un servlet** es bien distinto.

Veamos las características más importantes de la vida de un *servlet*.

- El *servlet* queda alojado en la memoria del servidor la primera vez que un usuario accede a él. A partir de ese momento, queda disponible para todos los usuarios que accedan a él.
- Miles de usuarios al mismo tiempo podrían estar solicitando la ejecución del *servlet* y, por lo tanto, provocando la ejecución de los métodos *doGet()* o *doPost()*, pero el *servlet* se ha tenido que registrar en el servidor una sola vez .
- El *servlet* seguirá estando activo y esperando solicitudes hasta que la persona encargada de la administración del servidor decida detenerlo por labores de mantenimiento.

En ocasiones, es necesario ejecutar una tarea solamente en el momento que se registra el *servlet* en el servidor; por ejemplo, para abrir conexión con una base de datos. Pues bien, es posible sobrescribir un método heredado de la clase *HttpServlet* para ese fin. Se trata del método *init()*. Por otro lado, también podemos sobrescribir el método *destroy()* para ejecutar alguna acción cuando el *servlet* pasa a liberarse de la memoria del servidor.

Modifica el código del primer *servlet* de la siguiente forma:

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/HolaMundo")
public class HolaMundo extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

```
public HolaMundo() {
    super();
}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter flujoEscritura=response.getWriter();

    flujoEscritura.append("<!DOCTYPE html>");
    flujoEscritura.append("<html><head><meta charset='UTF-8'>");
    flujoEscritura.append("<title>Página dinámica</title>");
    flujoEscritura.append("</head><body>");
    for (int i=1; i<7; i++) {
        flujoEscritura.append("<h"+i+">Hola Mundo</h"+i+">");
    }
    flujoEscritura.append("</body></html>");

    flujoEscritura.close();
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    doGet(request, response);
}

@Override
public void init() throws ServletException {
    System.out.println("El Servlet se ha registrado en el servidor");
}

@Override
public void destroy() {
    System.out.println("El Servlet ha sido retirado de la memoria del
servidor");
}

}
```

Hemos sobrescrito los métodos *init()* y *destroy()* mostrando un simple mensaje en la consola de Eclipse. Sólo verás el mensaje “El servlet se ha registrado en el servidor” la primera vez que se ejecute el *servlet* después de haber reiniciado el servidor.



Puede que te cueste más ver el mensaje mostrado por el método `destroy()`, ya que puede pasar tiempo hasta que el recolector de basura de Java libere realmente el objeto.

Acceso a datos desde un servlet

Ya has comprobado que **un servlet es en realidad una clase Java cualquiera, pero que extiende de HttpServlet**.

Recuerda que un *servlet* es una clase, y que desde él puedes acceder a todos los recursos de acceso a datos que has aprendido en este módulo.

Dentro de un servlet puedes:

- Acceder a **ficheros**, utilizando las clases que representan flujos de datos y que aprendiste en la unidad 1 de este módulo.
- Acceder a una **base de datos** relacional, utilizando el **API JDBC** tal y como aprendiste en la unidad 2 de este módulo.
- Acceder a una base de datos relacional utilizando un **framework ORM**, tal y como aprendiste en la unidad 3 de este módulo.
- Acceder a documentos **XML**, utilizando el **parser DOM** o solicitando los servicios de acceso a bases de datos XML que proporciona la aplicación **BaseX**.

Servlet con acceso a base de datos XML

Vamos a crear un *servlet* que genere una página dinámica con el inventario de artículos de la base de datos XML de ALMACEN que creamos en la unidad anterior.

Comienza por crear un proyecto web dinámico que se ejecutará en el servidor Apache Tomcat, tal y como has aprendido.

Dentro del proyecto, accederemos al servicio que presta BaseX para acceso a las bases de datos que administra, por lo que es importante que prepares el proyecto para dicho objetivo. Debes realizar un par de tareas para que tu proyecto funcione:

- Importa dentro del proyecto los tres archivos que conforman la librería BaseX XQJ API.
- **IMPORTANTE:** puesto que el proyecto será ejecutado por Apache Tomcat, copia también los tres archivos de la librería BaseX XQJ API dentro de la carpeta *lib* de la instalación de Apache Tomcat.

Dentro de tu nuevo proyecto, crea un *servlet* llamado *Inventario* con el siguiente código:

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;
import javax.xml.xquery.XQResultSequence;

import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import net.xqj.basex.BaseXXQDataSource;

@WebServlet("/Inventario")
public class Inventario extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Inventario() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        XQDataSource xds = new BaseXXQDataSource();
        XQConnection con;
        XQExpression expr;
        XQResultSequence result;
        String sentencia;
        String textoHTML = "";
    }
}
```

```

try {
    xds.setProperty("serverName", "localhost");
    xds.setProperty("port", "1984");
    con = xds.getConnection("admin", "admin");
    expr = con.createExpression();
    sentencia = "for $pro in fn:collection('almacen') //productos return
$pro/producto";
    result = expr.executeQuery(sentencia);
    while (result.next()) {
        Node nodoProducto = result.getNode();
        textoHTML = mostrarProducto(nodoProducto, textoHTML);
    }
    con.close();
} catch (XQException e) {
    textoHTML = "Error al obtener los datos XML <br />";
    textoHTML = textoHTML + e.getMessage();
}

response.setContentType("text/html");
PrintWriter flujoEscritura=response.getWriter();

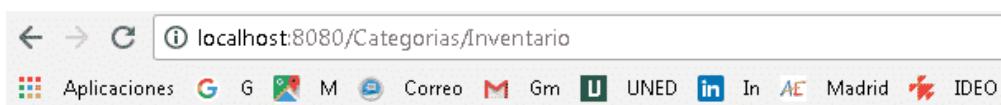
flujoEscritura.append("<!DOCTYPE html>");
flujoEscritura.append("<html><head><meta charset='UTF-8'>");
flujoEscritura.append("<title>Página dinámica</title>");
flujoEscritura.append("</head><body>");
flujoEscritura.append("<h1>Artículos de alimentación</h1>");
// Creamos una tabla HTML
flujoEscritura.append("<table>");
// Creamos la cabecera
flujoEscritura.append("<tr>");
flujoEscritura.append("<th>Artículo</th>");
flujoEscritura.append("<th>Cantidad por cada unidad</th>");
flujoEscritura.append("<th>Precio Unidad</th>");
flujoEscritura.append("<th>Stock</th>");
flujoEscritura.append("</tr>");
flujoEscritura.append(textoHTML);
flujoEscritura.append("</table>");
flujoEscritura.append("</body></html>");
flujoEscritura.close();
}

private static String mostrarProducto(Node nodo, String texto) {
    NodeList nodos = nodo.getChildNodes();
    // Creamos una fila HTML (Table Row)
    texto = texto + "<tr>";
    for (int i=0; i<nodos.getLength();i++) {
        Node nodoHijo = nodos.item(i);
        if (nodoHijo.getNodeType() == Node.ELEMENT_NODE) {
            // Creamos una celda de datos HTML (Table data)
            texto = texto + "<td>";
            texto = texto + nodoHijo.getTextContent();
            texto = texto + "</td>";
        }
    }
    texto = texto + "<tr>";
    return texto;
}
}

```

Si te fijas detenidamente en el código del *servlet*, verás que no hemos introducido nada nuevo, sino que hemos generado una página dinámica que **accede a BaseX para obtener el inventario de productos del almacén, itera los elementos *producto* obtenido y genera una tabla HTML como respuesta al cliente.**

El resultado en el navegador quedará así:



Artículos de alimentación

Artículo	Cantidad por cada unidad	Precio	Unidad	Stock
Té Dharamsala	10 cajas x 20 bolsas	20.16	39	
Cerveza tibetana Barley	24 - bot. 12 l	21.28	17	
Refresco Guaraná Fantástica	12 - latas 355 ml	5.04	20	
Cerveza Sasquatch	24 - bot. 12 l	15.68	111	
Cerveza negra Steeleye	24 - bot. 12 l	20.16	20	
Vino Côte de Blaye	12 - bot. 75 cl	295.12	17	
Licor verde Chartreuse	750 cc por bot.	20.16	69	
Café de Malasia	16 - latas 500 g	51.52	17	
Cerveza Laughing Lumberjack	24 - bot. 12 l	15.68	52	
Cerveza Outback	24 - bot. 355 ml	16.8	15	
Cerveza Klosterbier Rhönbräu	24 - bot. 0,5 l	8.68	125	
Licor Cloudberry	500 ml	20.16	57	
Sirope de regaliz	12 - bot. 550 ml	11.2	13	
Especias Cajun del chef Anton	48 - frascos 6 l	24.64	53	
Mezcla Gumbo del chef Anton	36 cajas	23.912	0	
Mermelada de grosellas de la abuela	12 - frascos 8 l	28	120	
Salsa de arándanos Northwoods	12 - frascos 12 l	44.8	6	
Salsa de soja baja en sodio	24 - bot. 250 ml	17.36	39	

Java Server Pages (JSP)

Introducción

Ahora aprenderás a crear **páginas dinámicas**, utilizando la tecnología Java Server Pages.

Java Server Pages es una tecnología que permite mezclar el código HTML, que se ejecuta en el *front-end*, con *script* de código Java que se ejecuta en el servidor y que forma parte del *backend*, todo esto en un único archivo con extensión *.jsp*. Tan sólo se debe escribir el código HTML de la manera habitual y luego encerrar el código Java de las partes dinámicas en etiquetas especiales, muchas de las cuales comienzan con **<%** y finalizan con **%>**.

Los **elementos básicos de cualquier página JSP** son:

- **Expresiones <%= expresión %>**
- **Scriptlets <% código java %>**
- **Declaraciones <%! código java %>**

Sigue adelante para profundizar más sobre estos elementos.

Elementos básicos de los documentos JSP

En este apartado vamos a conocer algunos **elementos básicos de los documentos JSP**.

1. Expresiones <%= expresión %>

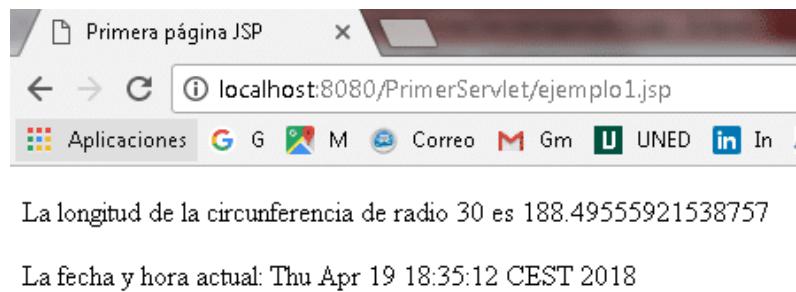
La expresión especificada se resuelve y su resultado se integra en el contenido HTML. Puedes ponerlo en práctica creando un archivo llamado *ejemplo1.jsp* con el siguiente contenido:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Primera página JSP </title>
</head>
<body>
    <p>La longitud de la circunferencia de radio 30 es
    <%= 2*30*java.lang.Math.PI%> </p>
    <p>La fecha y hora actual:
    <%= new java.util.Date()%> </p>
</html>
```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

Esta cabecera es una **directiva común a todas las páginas JSP**.

Cuando veas el resultado en el navegador, comprobarás que las expresiones se han resuelto: lo que ve el usuario es el resultado.



Prueba a hacer clic derecho desde el navegador y selecciona *Ver código fuente* para ver el código HTML. Comprobarás que el único contenido que aparece es HTML. Es algo así:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Primera página JSP </title>
</head>
<body>
    <p>La longitud de la circunferencia de radio 30 es
    188.49555921538757 </p>
    <p>La fecha y hora actual:
    Thu Apr 19 18:27:12 CEST 2018 </p>
</body>
</html>
```

La página JSP ha sido ejecutada en el servidor, que ha convertido las partes de código Java en código HTML estático. Este resultado estático es el que devuelve al cliente para que sea visualizado en el navegador.

2. Scriptlets <% código java %>

Un **scriptlet** es un bloque de código Java que es ejecutado en el servidor. Veamos un ejemplo: Crea otro archivo; esta vez, puedes llamarlo *ejemplo2.jsp*.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Articulo </title>
</head>
<body>
    <h1>INFORMACION SOBRE EL PRODUCTO</h1>
    <%
        double precio=10;
        int cantidad=5;
```

```

        String producto="Peras";
        double subtotal=precio*cantidad;
        if (subtotal>40) {
            subtotal = subtotal - (subtotal * 0.1);
        }
        double iva=subtotal*0.16;
        double total=subtotal+iva;
    %>
    <p>Nombre artículo:<%= producto %> </p>
    <p>Precio artículo:<%= precio %> </p>
    <p>Subtotal: <%= subtotal%> </p>
    <p>Total IVA incluido: <%= total %> </p>
</body>
</html>

```

El resultado en el navegador es:



3. Declaraciones <%! código java %>

Nos referimos a la declaración de variables o funciones cuyo ámbito y vigencia es global, hasta que el servidor se detiene por labores de mantenimiento.

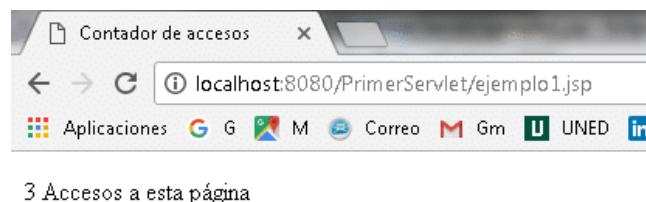
Crea ahora el archivo *ejemplo3.jsp* con el siguiente contenido:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Contador de accesos </title>
</head>
<body>
    <%!
        int contador=0;
    %>
    <% contador++;      %>
    <p><%= contador %> Accesos a esta página</p>
</body>
</html>

```

La variable *contador* está declarada dentro de una etiqueta especial JSP de tipo *declaración*, lo que permite que su ámbito sea global y se mantenga durante todo el tiempo que el servidor esté abierto. Puedes abrir la página incluso desde navegadores distintos y comprobar que se mantiene el valor anterior del contador.



Las variables predefinidas request y response

Para el servidor, una página JSP es como un *servlet*, ya que igualmente recibe peticiones de clientes y envía respuestas en formato HTML.

Cada página JSP es registrada en el servidor como un *servlet* y **tiene predefinidas las variables request y response**. Podríamos resolver el problema de los triángulos con una página JSP en sustitución del *servlet* anterior. Veamos cómo:

Crea la página ***triang.jsp*** con el siguiente código:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>Contador de accesos </title>
</head>
<body>
<%
    int lado1=0, lado2=0, lado3=0;
    String resultado;

    response.setContentType("text/html");

    try {
        lado1 = Integer.parseInt(request.getParameter("lado1"));
        lado2 = Integer.parseInt(request.getParameter("lado2"));
        lado3 = Integer.parseInt(request.getParameter("lado3"));

        if (lado1==lado2 && lado2==lado3)
            resultado = "EQUILÁTERO";
        else if (lado1==lado2 || lado2==lado3 || lado1==lado3)
            resultado = "ISÓSCELES";
        else
            resultado = "ESCALENO";

    } catch (NumberFormatException e) {
        resultado = "Alguno de los lados que has introducido no es correcto";
    }

%>
<h2>Triangulo con lados <%=lado1 %>, <%=lado2 %>, <%=lado3 %></h2>
<h2><%=resultado %></h2>
</body>
</html>
```

Observa que podemos seguir utilizando el objeto *request* para recoger los parámetros introducidos por el usuario. Los objetos *response* y *request* están predefinidos, y siempre disponibles para las páginas JSP.

Ahora, sólo te falta modificar la cabecera del formulario de la página *index.html* para que utilice la página JSP en lugar del *servlet*.

```
<form action="triang.jsp" method="get">
```

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Los **servlets** son programas que se ejecutan en un servidor web, en una capa intermedia entre una petición proveniente de un navegador web o cliente, y las bases de datos o aplicaciones del servidor HTTP. Su misión principal es **atender peticiones de clientes, leer los parámetros de entrada y enviar una página de respuesta.**
- **Java Server Pages** es una tecnología que permite mezclar el código HTML que se ejecuta en el *front-end*, con *script* de código Java que se ejecuta en el servidor y forma parte del *back-end*, todo esto en un único archivo con extensión *.jsp*. Tan sólo se debe escribir el código HTML de la manera habitual y luego encerrar el código Java de las partes dinámicas en etiquetas especiales, muchas de las cuales **comienzan con <% y finalizan con %>**.

5.6. Los JavaBeans



Índice

Objetivos	3
Los JavaBeans	4
Concepto	4
Proyecto web con un JavaBean.....	5
JSP y los JavaBeans	7
Las directivas <i>useBean</i> y <i>setProperty</i> de JSP	8
Añadir Scripts y expresiones JSP.....	9
Despedida	11
Resumen.....	11

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Recordar el concepto de *JavaBeans*.
- Utilizar *JavaBeans* desde páginas JSP.

Los JavaBeans

Concepto

En el siguiente apartado vamos a recordar el concepto de **JavaBeans**, que ya vimos en la Unidad 3, con el objeto de aplicarlo después a las páginas JSP.

Para comenzar, recordemos que un *JavaBean* es una **clase** que debe cumplir una serie de condiciones.

- Debe implementar la **interfaz Serializable**, que otorga a sus objetos la capacidad de persistencia.
- Debe tener un **constructor vacío** (que no reciba argumentos), aunque luego puede tener otros constructores. De esta forma, permite crear objetos estándar.
- Todas las propiedades del objeto serán **privadas y accesibles mediante métodos get/set** que serán públicos.
- Para los métodos *get/set*, hay que **seguir cuidadosamente la nomenclatura estándar**. Para una propiedad privada llamada *precio*, los métodos *get/set* serán *getPrecio* y *setPrecio*, es decir, con las palabras *get* y *set* en minúscula y el nombre de la propiedad con la primera letra mayúscula.

El concepto *JavaBean* fue creado por Sun Microsystems, aunque esta compañía fue adquirida por Oracle en 2010. Los *JavaBean* nacieron con el objetivo de ser utilizados como componentes software reutilizables.

Ejemplo:

Veamos un ejemplo de clase que cumple con las especificaciones de los *JavaBeans*.

```
import java.io.Serializable;
import java.time.LocalDateTime;

public class Llamada implements Serializable {
    private static final long serialVersionUID = 6164080316086841480L;

    private LocalDateTime fechaHora;
    private String emisor; // Nombre de la persona que llamo.
    private String motivo; // Motivo de la llamada.

    public Llamada() {
        this.fechaHora = LocalDateTime.now();
    }

    public LocalDateTime getFechaHora() {
        return fechaHora;
    }
    public void setFechaHora(LocalDateTime fechaHora) {
        this.fechaHora = fechaHora;
    }
    public String getEmisor() {
```

```
        return emisor;
    }
    public void setEmisor(String emisor) {
        this.emisor = emisor;
    }
    public String getMotivo() {
        return motivo;
    }
    public void setMotivo(String motivo) {
        this.motivo = motivo;
    }

    @Override
    public String toString() {
        return this.emisor + " llamo el " + this.fechaHora + " para " +
this.motivo;
    }
}
```

La clase *Llamada* cumple con la especificación *JavaBeans* porque es serializable, tiene un constructor sin argumentos, y sus métodos son privados y accesibles mediante métodos *get/set*, generados cumpliendo la nomenclatura estándar.

Podríamos construir un objeto en una clase con método *main()* de la siguiente forma:

```
public class Principal {

    public static void main(String[] args) {
        Llamada unaLlamada = new Llamada();
        unaLlamada.setEmisor("Carlos Pérez");
        unaLlamada.setMotivo("Pedir información");
        System.out.println(unaLlamada); // Invoca al método toString();
    }
}
```

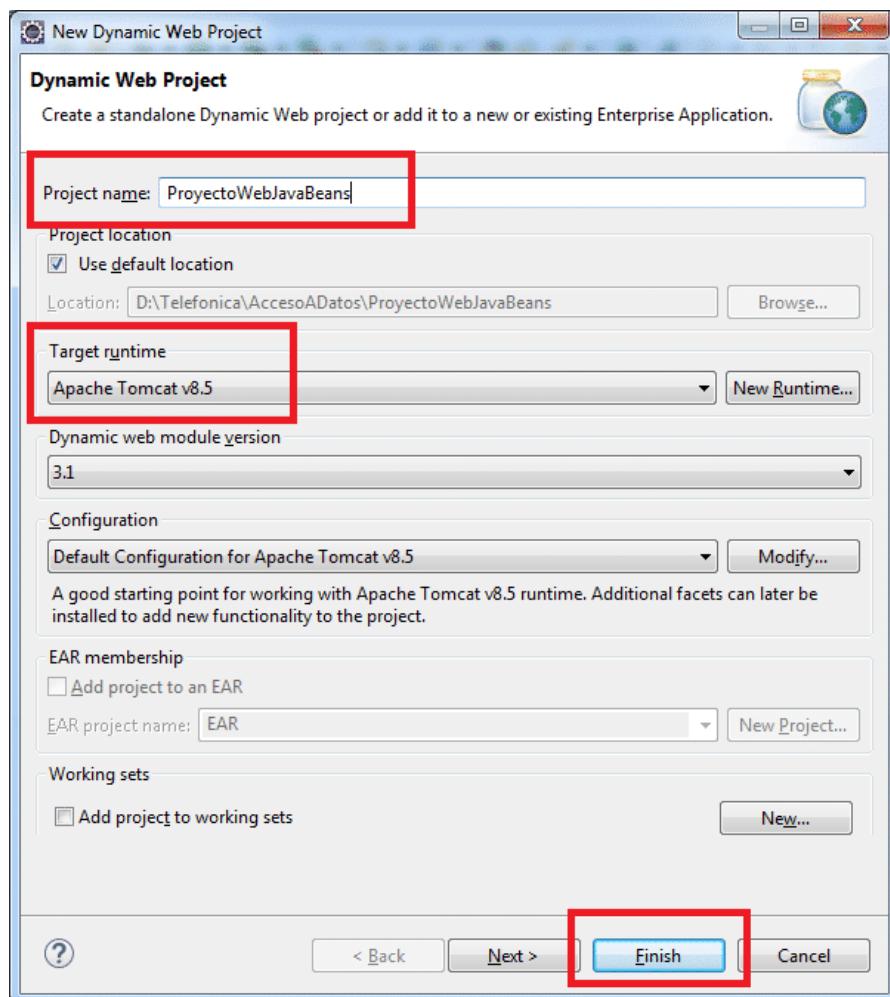
Proyecto web con un JavaBean

En este apartado, crearemos un proyecto web dinámico con un *JavaBean* para después utilizarlo dentro de una página JSP.

Sigue estos pasos:

1. Crea un proyecto web dinámico llamado *ProyectoWebJavaBeans*

Sitúate en la perspectiva Java EE, si es que no estás ya en ella. Despues, selecciona en el menú **File / New / Dynamic Web Project**. Recuerda que debes especificar el nombre del proyecto y el *Target runtime*, apuntando a la ubicación de la instalación de Apache Tomcat que tengas en tu equipo.



2. Crea la clase que servirá de *JavaBean*

Vamos a crear una clase llamada *Estudiante* en un paquete llamado *escuela*. Como para cualquier otro tipo de clase, haz clic derecho sobre el nombre del proyecto y selecciona *New / Class*.

Luego, edita el código hasta dejarlo así:

```
package escuela;

import java.io.Serializable;
import java.time.LocalDateTime;

public class Estudiante implements Serializable {
    private static final long serialVersionUID = 7556602068002620452L;

    String nombre;
    String curso;
    LocalDateTime fechaHoraMatricula;

    public Estudiante() {
        this.nombre = "Desconocido";
        this.curso = "Java";
        this.fechaHoraMatricula = LocalDateTime.now();
```

```
}

public String getNombre() {
    return nombre;
}

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

public String getCurso() {
    return curso;
}

public void setCurso(String curso) {
    this.curso = curso;
}

public LocalDateTime getFechaHoraMatricula() {
    return fechaHoraMatricula;
}

public void setFechaHoraMatricula(LocalDateTime fechaHoraMatricula) {
    this.fechaHoraMatricula = fechaHoraMatricula;
}

}
```

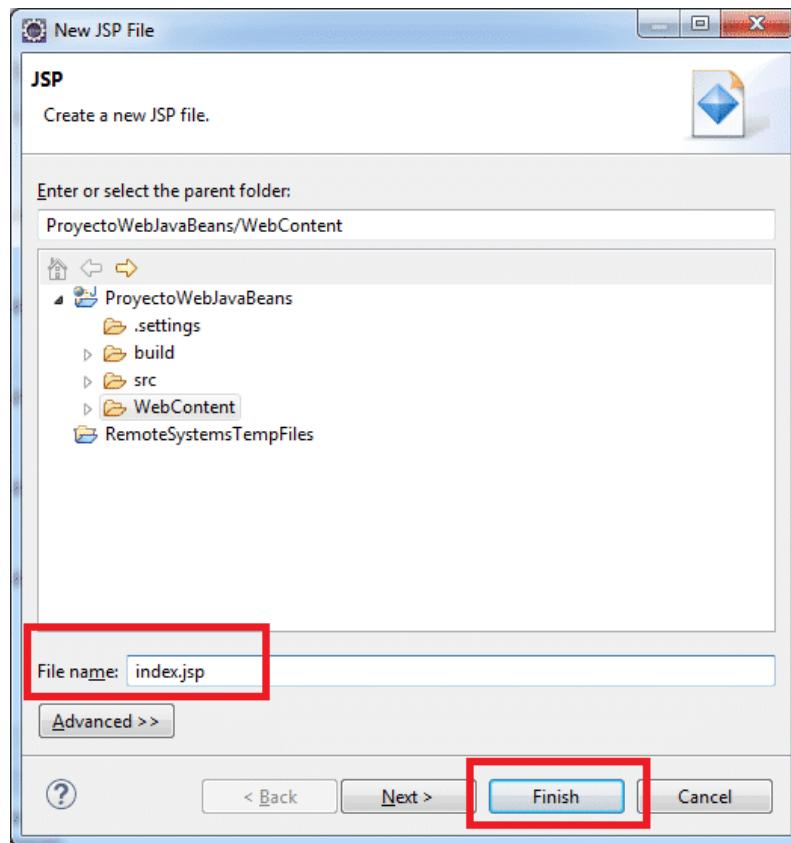
Como puedes comprobar, la clase **Estudiante** cumple todos los requisitos para ser un **JavaBean**.

JSP y los JavaBeans

En el siguiente apartado, vamos a hacer uso de la clase **Estudiante** dentro de una página dinámica.

1. Comienza por crear tu archivo JSP.

Haz clic derecho sobre el nombre del proyecto y selecciona New / JSP File en el menú contextual. El nombre de la página será *index.jsp*.



2. Sustituye el código que aparece por defecto por el que te presentamos a continuación:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<jsp:useBean id="yo" class="escuela.Estudiante" />
<jsp:setProperty name="yo" property="nombre" value="Amelia González" />
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Página con un JavaBean</title>
</head>
<body>

</body>
</html>
```

Hemos aplicados dos directivas de JSP que no habías utilizado hasta ahora. Vamos a comprobarlo.

Las directivas *useBean* y *setProperty* de JSP

La **directiva *jsp:useBean*** permite construir un objeto a partir de una clase que cumpla con los requisitos de los *JavaBeans*. A los objetos creados de esta forma los denominamos ***beans***.

```
<jsp:useBean id="yo" class="escuela.Estudiante" />
```

La línea anterior es equivalente a una declaración Java de este tipo:

```
escuela.Estudiante yo = new Estudiante();
```

El atributo *id* es el que determinará el nombre del objeto.

La directiva ***jsp:setProperty*** se utiliza para asignar valores a las propiedades de un *bean*.

```
<jsp:setProperty name="yo" property="nombre" value="Amelia González" />
```

La línea anterior es equivalente a cuando en Java escribimos lo siguiente:

```
yo.setNombre("Amelia González");
```

Añadir Scripts y expresiones JSP

Recuerda que puedes incluir código Java dentro de un script JSP que va encerrado entre los símbolos `<% %>`. En esta ocasión, utilizaremos un script para asignar el valor *Acceso a datos* a la propiedad *curso*; de este modo, comprobarás que también podemos seguir utilizando el sistema clásico de Java.

Completa el código hasta dejarlo así:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<jsp:useBean id="yo" class="escuela.Estudiante" />
<jsp:setProperty name="yo" property="nombre" value="Perico de los Palotes" />
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Página con un JavaBean</title>
</head>
<body>
<%
    yo.setCurso("Acceso a datos");
%>
<h1>Estudiante: <%=yo.getNombre() %></h1>
<h2>Fecha/hora matrícula: <%=yo.getFechaHoraMatricula() %></h2>
<h3>Curso: <%=yo.getCurso() %></h3>
</body>
</html>
```

Dentro del código HTML hemos incluido expresiones JSP para mostrar los valores de las propiedades.

Recuerda que una expresión JSP va encerrada entre los símbolos <%= %>.

Ver el resultado en el navegador

Como sabes, puedes utilizar el botón *Run* dentro de Eclipse para ver cómo quedará la página *index.jsp* en el navegador.



Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Un **JavaBean** es una clase que debe cumplir una serie de condiciones: implementar la interfaz *Serializable*, tener un constructor vacío y propiedades privadas accesibles mediante métodos *get/set*.
- Por medio de la directiva **useBean** podemos construir un objeto a partir de una clase que cumpla con la especificación de los *JavaBean* dentro de una página JSP. Ejemplo:
`<jsp:useBean id="yo" class="escuela.Estudiante" />`
- Por medio de la directiva **setProperty** podemos asignar valores a las propiedades de un *bean* (objeto creado con *useBean* dentro de una página JSP).
`<jsp:setProperty name="yo" property="nombre" value="Perico de los Palotes" />`

6.1. Características de las bases de datos orientadas a objetos



Índice

Objetivos	3
Modelos de bases de datos	4
Bases de datos relacionales	4
Bases de datos orientadas a objetos	5
Bases de datos objeto-relacionales.....	7
Despedida	8
Resumen.....	8

Objetivos

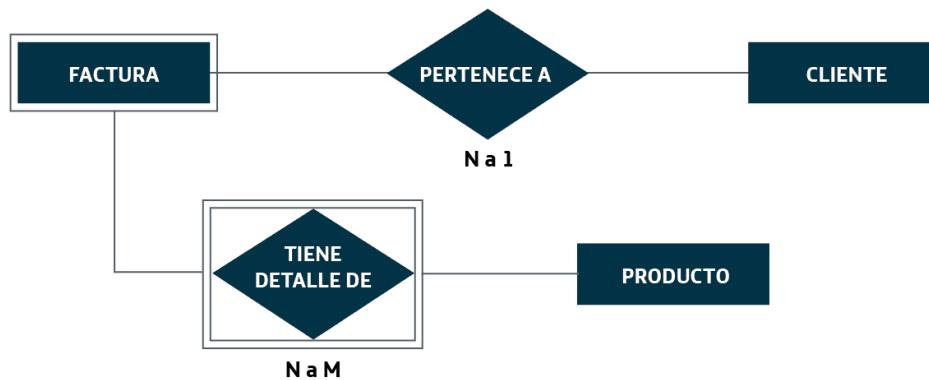
En esta lección perseguimos los siguientes objetivos:

- Conocer los fundamentos de las bases de datos orientadas a objetos.
- Distinguir entre base de datos relacional, base de datos orientada a objetos y base de datos objeto relacional.

Modelos de bases de datos

Bases de datos relacionales

Una base de datos que sigue el modelo relacional está formada por un conjunto de tablas o relaciones de datos que están interconectadas entre sí por medio de asociaciones.



Nuestra base de datos *FERRETERIA* tiene una tabla *CLIENTE* interconectada con otra tabla *FACTURA* a través del *NIF*.

Cada tabla está formada por un conjunto de filas, donde cada una de ellas almacena información sobre una determinada entidad, por ejemplo: un producto. Cada fila está compuesta por atributos cuyos valores sólo pueden ser de tipo elemental o atómico, es decir, que no podrá descomponerse en varios valores.

En las bases de datos relacionales, la información necesaria para elaborar un documento debe ser extraída normalmente de varios registros de varias tablas diferentes. Por ejemplo: para elaborar una factura, será necesario extraer los datos del cliente de la tabla *CLIENTE*, los datos factura de la tabla *FACTURA*, las ventas de la tabla *DETALLE* y las descripciones de los artículos de la tabla *PRODUCTO*. **Este detalle es lo que diferencia a las bases de datos relacionales de las bases de datos documentales**, en las que la información necesaria para confeccionar un documento está almacenada en una única unidad o registro.

En los siguientes apartados podrás comparar el modelo relacional de base de datos con el modelo orientado a objetos y el modelo objeto-relacional.

Bases de datos orientadas a objetos

Un sistema de gestión de bases de datos orientado a objetos no es más que un DBMS que soporta el paradigma de orientación a objetos.

En la primera unidad de esta módulo, aprendiste a guardar objetos en disco. En concreto, creaste un objeto *Alumno* con su colección de objetos *Calificacion* y lo guardaste en un fichero. También construiste una pequeña agenda telefónica como un archivo que almacena objetos *Contacto*. Aquello fue la base para comprender el funcionamiento de los sistemas de gestión de bases de datos orientadas a objetos.

Cuando hablamos de gestores de bases de datos orientadas a objetos, hablamos de herramientas cuya principal labor consiste en facilitar la **persistencia de objetos**, es decir, guardar colecciones de objetos en un dispositivo de almacenamiento con el fin de recuperarlos cuando sea necesario. Recuerda que para que un objeto tenga la capacidad de persistencia, debe pertenecer a una clase que implemente la interfaz *Serializable*.

Te recomendamos que repases la lección **1.2. Lectura/Escritura de objetos** de la primera Unidad Formativa de esta asignatura.

Los DBMS orientados a objetos proveen de un lenguaje basado en SQL, pero que permite la recuperación del subconjunto de objetos que cumplan un criterio dado dentro de la colección completa de objetos guardados en la base de datos. Este sublenguaje se denomina **OQL (Object Query Language)**.

Veamos un ejemplo de consulta en SQL y su correspondencia en OQL:

1. SQL:

```
SELECT descripcion, precio FROM Producto  
WHERE codigo = 'TOR7';
```

En este caso, el DBMS nos ha devuelto una fila o registro compuesto por los atributos atómicos *descripcion* y *precio*, que no podrán descomponerse en nada más.

2. OQL:

```
SELECT pr.descripcion, pr.precio FROM pr IN Producto  
WHERE pr.codigo = 'TOR7';
```

En este caso, el DBMS nos devuelve un objeto de la clase *Producto*, cuya referencia es *pr* y cuyos atributos *descripcion* y *precio* pueden ser atómicos pero también podrían ser objetos que se descomponen en otros atributos (composición).

3. OQL con un atributo que se descompone:

```
SELECT cl.nombre, cl.direccion, cl.telefono
FROM cl IN Cliente
WHERE cl.direccion.codigoPostal = '28017';
```

En este otro ejemplo estamos recuperando una colección de objetos *Cliente* cuyo *codigoPostal* tenga el valor '28017'. Cada objeto *Cliente* está compuesto por *nombre*, *direccion* y *telefono*, pero la *direccion*, a su vez, es otro objeto compuesto por *calle*, *numero*, *piso*, *puerta*, *codigoPostal*, etc.

Tres ejemplos de ODBMS (Object Data Base Management System):

ZOPE OBJECT DATABASE

Se trata de un ODBMS (*Object Data Base Management System*) creado por Zope Corporation a finales de los 90. Permite el almacenamiento y recuperación de objetos en lenguaje Python. Incluye transacciones, historial y deshacer, concurrencia y escalabilidad.

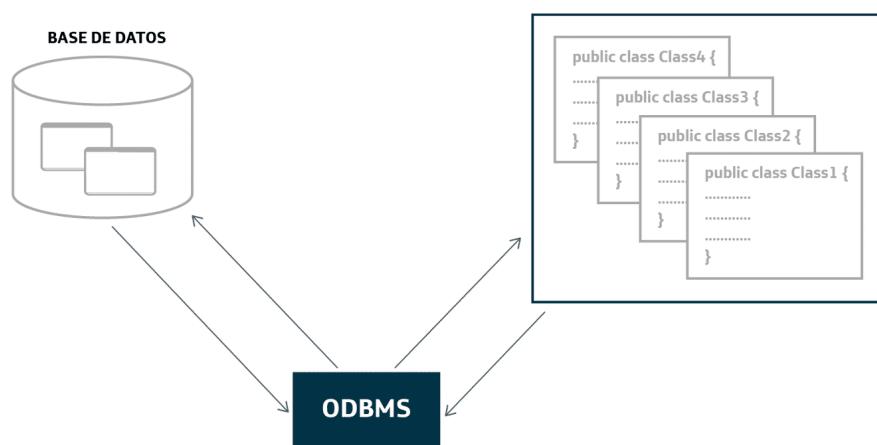
OBJECTDB

Se trata de un ODBMS muy orientado para interactuar con programas Java. Sin embargo, no dispone de API propietaria, por lo que es necesario utilizar alguna API de las estándar de Java, como, por ejemplo, JPA.

NEODATIS

Se trata de un ODBMS muy simple, escrito en Java y que cuenta con su propia API propietaria que puede ser importada en cualquier aplicación Java.

Éste podría ser el esquema de funcionamiento de un ODBMS:



La base de datos almacena objetos y el ODBMS se encarga de su administración y de suministrar los objetos requeridos por los programas que interactúan con él.

Las bases de datos orientadas a objetos pertenecen a la categoría de **bases de datos documentales**, donde toda la información necesaria para confeccionar un documento, como una factura, se encuentra disponible en una única unidad o registro; en este caso, en un único objeto, aunque luego dicho objeto nos da acceso a otros objetos dentro de su jerarquía.

Bases de datos objeto-relacionales

Los DBMS objeto-relacionales se quedan a medio camino entre el modelo relacional y el modelo orientado a objetos, utilizando lo mejor de ambos.

Las bases de datos objeto-relacionales siguen utilizando la estructura relacional basada en tablas con filas y columnas, donde cada tabla tiene asociaciones con otras tablas. La diferencia radica en que cada atributo o campo dentro de un registro o fila puede ser, bien atómico (de tipo elemental), bien un tipo de dato más complejo.

Cada atributo puede ser de uno de estos tipos:

- **Tipo de dato elemental** (*INT, CHAR, FLOAT, etc.*).
- **Tipo estructurado** formado por más atributos. Por ejemplo: dirección compuesta por calle, número, piso, etc.
- **Objetos de gran tamaño**: canciones, imágenes, vídeos, etc.

Además, los DBMS objeto-relacionales poseen características de la programación orientada a objetos, como, por ejemplo, la definición de tipos personalizados (clases) y la herencia.

Dada la declaración del tipo estructurado *Persona*:

```
CREATE TYPE Persona  
(nombre VARCHAR(20), apellidos VARCHAR(20), tlf VARCHAR(10))
```

Podemos crear otro tipo que herede de *Persona*:

```
CREATE TYPE Alumno UNDER Persona (curso VARCHAR(20));
```

El tipo estructurado *Alumno* dispone del atributo *curso*, además de los atributos *nombre*, *apellidos* y *tlf*, que ha heredado de *Persona*.

Sin duda, Oracle Database es actualmente el gestor de bases de datos objeto-relacionales más importante.

Para saber más

Consulta la entrada de la wikipedia sobre Oracle Database.

https://es.wikipedia.org/wiki/Wikipedia:Comunicado_4_julio_2018

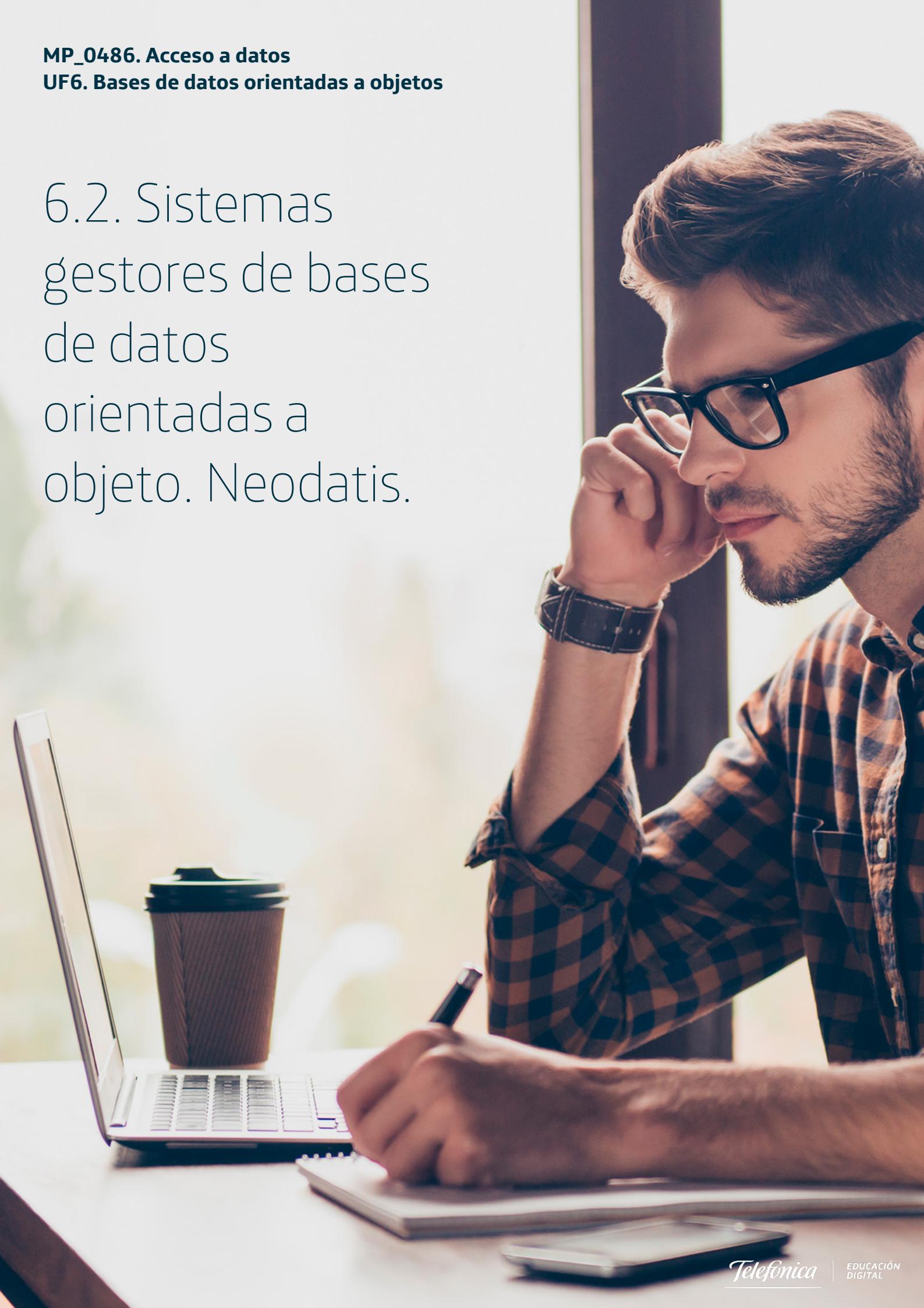
Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Una base de datos que sigue el **modelo relacional** está formada por un conjunto de tablas o relaciones de datos, que están interconectadas entre sí por medio de asociaciones. Cada tabla está formada por un conjunto de filas, donde cada una de ellas almacena información sobre una determinada entidad, por ejemplo, un producto. Cada fila está compuesta por atributos cuyos valores sólo pueden ser de tipo elemental o atómico, es decir, que no podrá descomponerse en varios valores.
- Un sistema de gestión de **bases de datos orientado a objetos** no es más que un DBMS que soporta el paradigma de orientación a objetos. Son herramientas cuya principal labor consiste en facilitar la persistencia de objetos, es decir, guardar colecciones de objetos en un dispositivo de almacenamiento con el fin de recuperarlos cuando sea necesario. Además, proveen de un lenguaje basado en SQL, pero que permite la recuperación del subconjunto de objetos que cumplan un criterio dado, dentro de la colección completa de objetos guardados en la base de datos. Este sublenguaje se denomina OQL (*Object Query Language*).
- Los **DBMS objeto-relacionales** se quedan en a medio camino entre el modelo relacional y el modelo orientado a objetos, utilizando lo mejor de ambos. Las bases de datos objeto-relacionales siguen utilizando la estructura relacional basada en tablas con filas y columnas, donde cada tabla tiene asociaciones con otras tablas. La diferencia radica en que cada atributo o campo dentro de un registro o fila puede ser atómico (de tipo elemental), o bien un tipo de dato más complejo.

6.2. Sistemas gestores de bases de datos orientadas a objeto. Neodatis.



Índice

Objetivos	3
Neodatis	4
Introducción.....	4
Obtener NeoDatis	4
Preparar el proyecto JAVA.....	6
Crear el modelo de clases: <i>Alumno</i> y <i>Calificacion</i>	6
Importar la librería de NeoDatis	7
Despedida	9
Resumen.....	9

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Conocer las características de Neodatis.
- Descargar el ODBMS Neodatis.
- Preparar un proyecto Java para interactuar con Neodatis.

Neodatis

Introducción

NeoDatis Object Database es un **gestor de base de datos orientada a objetos libre**, es decir, que sus usuarios tienen plena libertad para copiar, ejecutar, distribuir, estudiar, modificar y mejorar el código fuente.



Este gestor de bases de datos está íntegramente escrito en Java y ubicado en un único fichero .jar, que podemos incluir dentro de nuestros proyectos Java para enriquecerlos con el uso de las clases que aporta para el tratamiento de las bases de datos.

Utilizando la librería de clases que nos brinda NeoDatis dentro de nuestros proyectos, tendríamos ya implementada la capa de persistencia para guardar y recuperar objetos con una sola línea de código. De esta forma, podemos enfocar nuestro trabajo en la lógica de negocio.

La librería de NeoDatis puede integrarse en lenguajes como Java, .Net o Google Android.

NeoDatis también cuenta con una interfaz de usuario que permite examinar las bases de datos y también admite la importación / exportación a formato XML.

Obtener NeoDatis

En este apartado **descargarás y descomprimirás la aplicación NeoDatis**.

Sigue estos pasos para obtener NeoDatis y ejecutar la aplicación:

1. Utiliza un navegador web para acceder a la dirección <https://sourceforge.net/projects/neodatis-odb/> y haz clic en el enlace *Download*.

A screenshot of the SourceForge project page for NeoDatis ODB. At the top, there's a navigation bar with links for Articles, Cloud Storage, Business VoIP, and Internet Speed Test. Below the navigation is a banner featuring a photo of seafood. The main content area shows the project name "NeoDatis ODB" with a sub-link "Brought to you by: olivier_smadja". It displays a rating of 5 stars from 16 reviews and shows "Downloads: 2 This Week". A prominent green "Download" button is highlighted with a red border. At the bottom of the page, there are links for "VER FICHA", "MÍS...", and "LEER".

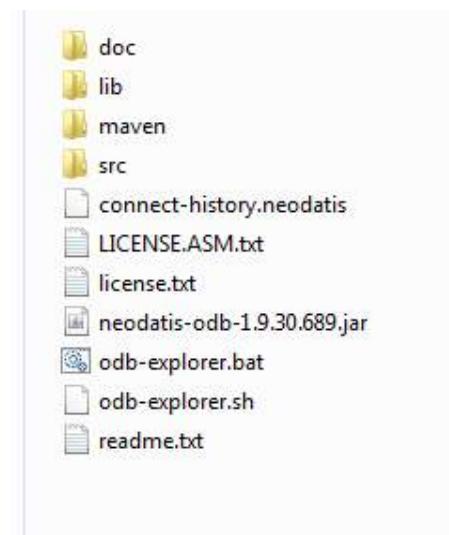
2. Espera a que termine la descarga del archivo y, después, guárdalo en la ubicación que deseas. Si no lo guardaste bien, seguro que lo encontrarás en la carpeta *Descargas* de Windows. El nombre del archivo que has obtenido debe ser ***neodatis-odb-1.9.30.689.zip***.

3. Como puedes comprobar, se trata de un archivo .zip que tendrás que descomprimir en la ubicación que deseas. Te aconsejo que lo descomprimas dentro de la carpeta donde guardas todos tus archivos del curso. Una vez que lo hayas descomprimido, habrás obtenido una carpeta con un nombre similar a este: ***neodatis-odb-1.9.30-689***.

4. A hora, abre la carpeta y fíjate en su contenido:

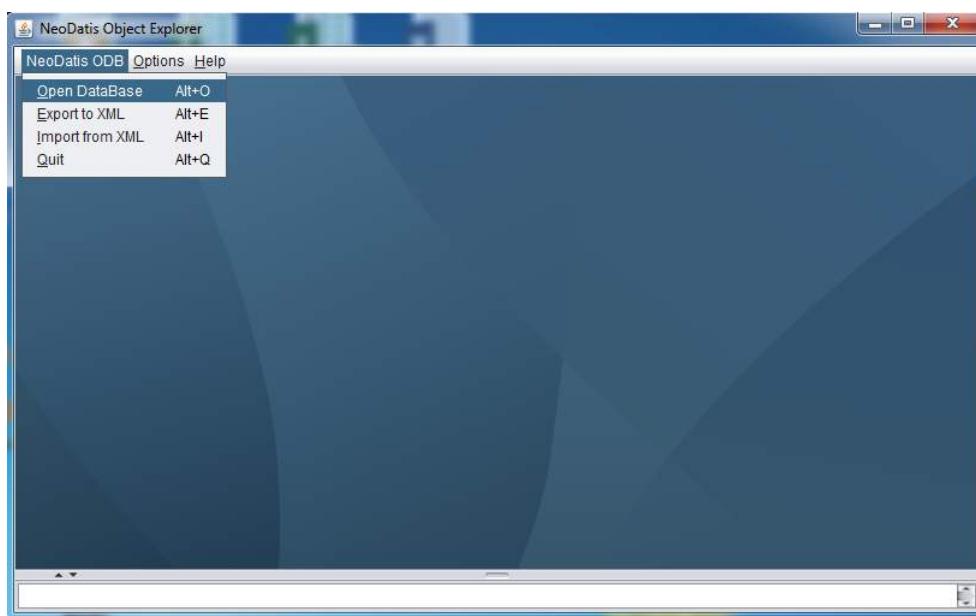
Lo único que nos interesa son un par de archivos que vamos a comentar a continuación:

- ***neodatis-odb-1.9.30.689.jar***: librería de clases que conforman la aplicación NeoDatis y que podemos importar dentro de nuestras aplicaciones Java, para interactuar con bases de datos orientadas a objetos.
- ***odb-explorer.bat***: archivo por lotes que lanza la interfaz de usuario de la aplicación NeoDatis. Desde esta interfaz de usuario podemos examinar las bases de datos orientadas a objetos que vayamos creando.



Haz doble clic sobre el archivo ***odb-explorer.bat*** para abrir NeoDatis por primera vez.

Éste es el aspecto que tendrá la interfaz de usuario de NeoDatis:



Interfaz de usuario NeoDatis.

Preparar el proyecto JAVA

Crear el modelo de clases: *Alumno* y *Calificacion*

Vamos a crear una **base de datos orientada a objetos para guardar la información de varios alumnos y sus calificaciones**. Para ello, en este apartado dejaremos preparado un proyecto Eclipse que nos servirá para interactuar con dicha base.

Volveremos a utilizar las clases *Alumno* y *Calificacion* que creaste en la unidad 1.2. Si no las tienes a mano no hay problema, crea un proyecto nuevo copiando y pegando las clases desde este apartado.

Dentro de Eclipse, crea un nuevo proyecto Java denominado *ProyectoODBMS* (*File / New / Java Project*).

Dentro del proyecto crea las clases *Alumno* y *Calificacion*, copiando los siguientes códigos:

```
import java.io.Serializable;
import java.util.ArrayList;

public class Alumno implements Serializable {
    private static final long serialVersionUID = 4854486451470258537L;

    private String nombre;
    private int edad;
    private ArrayList<Calificacion> calificaciones;

    public Alumno(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        this.calificaciones = new ArrayList<Calificacion>();
    }

    public void calificar(String asignatura, int nota) {
        this.calificaciones.add(new Calificacion(asignatura, nota));
    }

    public String getNombre() {
        return nombre;
    }
    public int getEdad() {
        return edad;
    }
    public ArrayList<Calificacion> getCalificaciones() {
        return calificaciones;
    }
}
```

Clase *Alumno*

```
import java.io.Serializable;

public class Calificacion implements Serializable {
    private static final long serialVersionUID = 3057545624874202352L;

    private String asignatura;
    private int nota; // Sobre 100

    public Calificacion(String asignatura, int nota) {
        this.asignatura = asignatura;
        this.nota = nota;
    }

    @Override
    public String toString() {
        return "Calificación [Asignatura=" + asignatura + ", Nota=" + nota + "]";
    }
}
```

Clase *Calificacion*

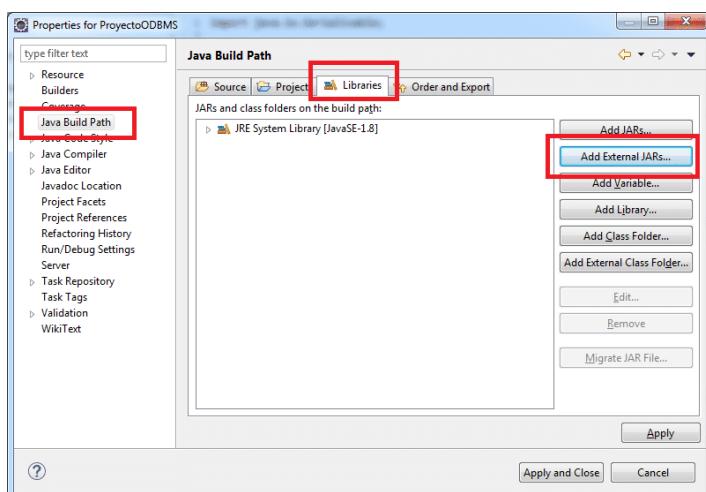
Ambas clases implementan la interfaz *Serializable*, lo que otorga a sus objetos la capacidad de persistencia.

Importar la librería de NeoDatis

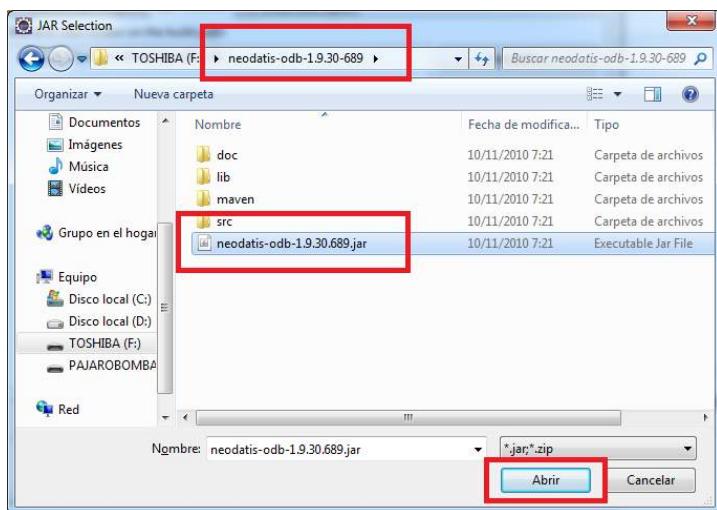
Ya tenemos el proyecto Eclipse con las clases *Alumno* y *Calificacion*. Ahora **vamos a importar el archivo .jar que contiene la librería de clases de NeoDatis**.

Sigue estos pasos:

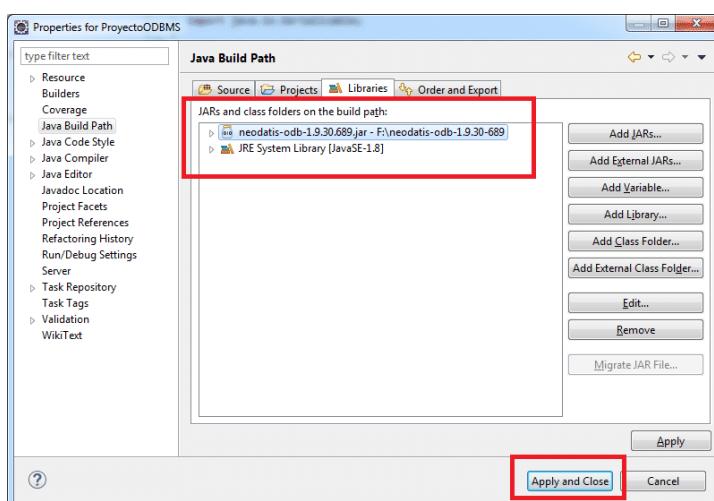
- 1. Haz clic derecho** en el nombre del proyecto y selecciona **Properties** en el menú contextual.
- 2. En el cuadro de diálogo Properties for ProyectoODBMS**, selecciona *Java Build Path*, activa la ficha **Libraries** y haz clic en el botón **Add External JARs...**.



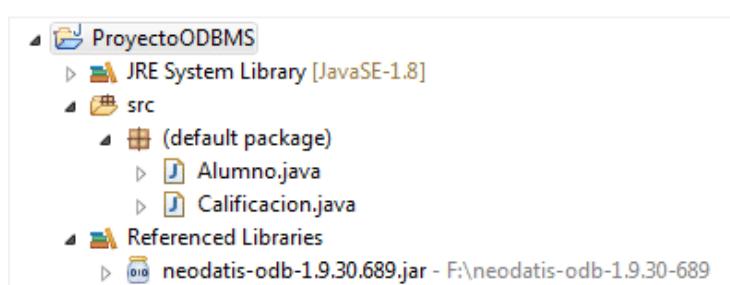
3. En el cuadro de diálogo **JAR Selection** busca la carpeta donde dejaste descomprimida la aplicación NeoDatis, selecciona el archivo **.jar** y haz clic en el botón **Abrir**, tal y como ves en la imagen.



El cuadro de diálogo *Properties for ProyectoODBMS* ya está mostrando la nueva librería incluida. Ahora ya sólo tienes que hacer clic en el botón **Apply and Close**.



Ahora mismo, tu proyecto tiene la siguiente estructura:



En la siguiente lección haremos uso de la librería que acabas de importar. Por ahora, nos conformamos con haber dejado configurado un proyecto para interactuar con NeoDatis.

Despedida

Resumen

Has terminado la lección, vamos a repasar los puntos más importantes que hemos tratado.

- NeoDatis Object Database es un **gestor de base de datos orientadas a objetos libre**.
- Está **escrito íntegramente en Java** y ubicado en un único archivo .jar.
- **Utilizando la librería de clases que nos brinda NeoDatis** dentro de nuestros proyectos, **tendremos ya implementada la capa de persistencia** para guardar y recuperar objetos con una sola línea de código. La librería de NeoDatis puede integrarse en lenguajes como Java, .Net o Google Android.
- NeoDatis también **cuenta con una interfaz de usuario** que permite examinar las bases de datos y también admite la importación / exportación a formato XML.

El interfaz de
programación de
aplicaciones de la
base de datos.
Librería Neodatis.



Índice

Objetivos	3
Crear BD Alumnos.....	4
Crear base de datos de alumnos en Java	4
Examinar base de datos desde NeoDatis	6
Distintas vistas para examinar la BD.....	8
Object View.....	9
Table View	10
Query	11
New Object	12
Consultar BD Alumnos	13
Consultar base de datos de alumnos en Java.....	13
Importar y exportar en Neodatis	15
Exportar Object DB a XML	15
Importar XML para obtener Object DB	17
Despedida	21
Resumen.....	21

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Desarrollar un programa Java capaz de crear una base de datos orientada a objetos, con ayuda de las clases de la librería de NeoDatis.
- Examinar los objetos almacenados en la base de datos desde la aplicación NeoDatis.
- Leer el contenido de una base de datos orientada a objetos desde Java, con ayuda de la librería NeoDatis.
- Exportar una base de datos orientada a objetos a formato XML.
- Importar un archivo XML para obtener una base de datos orientada a objetos.

Crear BD Alumnos

Crear base de datos de alumnos en Java

Ha llegado el momento de **crear la base de datos de alumnos con ayuda de la librería de clases de NeoDatis**, que importaste en la lección anterior.

Dentro del proyecto **ProyectoODBMS**, crea la clase **Principal**. Después, copia y pega el siguiente código:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;

public class Principal {

    public static void main(String[] args) {
        Alumno alu1 = new Alumno("Miguel", 25);
        Alumno alu2 = new Alumno("Pedro", 17);
        Alumno alu3 = new Alumno("Rosa", 19);

        alu1.calificar("Matemáticas", 45);
        alu1.calificar("Inglés", 70);
        alu1.calificar("TIC", 85);
        alu1.calificar("Física", 55);

        alu2.calificar("Matemáticas", 65);
        alu2.calificar("Inglés", 70);
        alu2.calificar("TIC", 95);
        alu2.calificar("Ciencias Naturales", 83);

        alu3.calificar("Matemáticas", 90);
        alu3.calificar("Inglés", 53);
        alu3.calificar("TIC", 75);
        alu3.calificar("Física", 25);

        ODB odbAlumnos = ODBFactory.open("G:/ALUMNOS.DB");
        odbAlumnos.store(alu1);
        odbAlumnos.store(alu2);
        odbAlumnos.store(alu3);

        odbAlumnos.close();
        System.out.println("Se ha creado la DBOO con Ayuda de NeoDatis");
    }
}
```

Vamos a analizar el código anterior paso a paso.

1. En primer lugar, creamos tres objetos de la clase **Alumno**.

Creamos tres objetos de la clase **Alumno** con sus calificaciones asociadas. Cada alumno se examina de cuatro asignaturas.

```
Alumno alu1 = new Alumno("Miguel", 25);
Alumno alu2 = new Alumno("Pedro", 17);
Alumno alu3 = new Alumno("Rosa", 19);
```

```
alu1.calificar("Matemáticas", 45);
alu1.calificar("Inglés", 70);
alu1.calificar("TIC", 85);
alu1.calificar("Física", 55);

alu2.calificar("Matemáticas", 65);
alu2.calificar("Inglés", 70);
alu2.calificar("TIC", 95);
alu2.calificar("Ciencias Naturales", 83);

alu3.calificar("Matemáticas", 90);
alu3.calificar("Inglés", 53);
alu3.calificar("TIC", 75);
alu3.calificar("Física", 25);
```

2. En segundo lugar, obtenemos la conexión con la base de datos *Alumnos*.

```
ODB odbAlumnos = ODBFactory.open("G:/ALUMNOS.DB");
```

El método **open** de la clase *ODBFactory* abre la base de datos orientada a objetos especificada en el fichero y retorna un objeto *ODB*, que representa dicha base de datos. Si no existe el fichero de base de datos especificado, será creado.

En el ejemplo hemos utilizado la ruta G:/ para guardar el archivo ALUMNOS.DB. Debes sustituir dicha ruta por la que deseas, dentro del sistema de archivos de tu equipo local.

Veamos más detenidamente las dos clases empleadas en la línea de código que nos ocupa:

- **ODBFactory**: esta clase es imprescindible, ya que actúa de puente entre la capa de persistencia, suministrada por la librería NeoDatis, y el resto de capas de nuestra aplicación Java. Además, nos permite abrir una base de datos orientada a objetos a través del método estático *open*, retornando un objeto de la clase *ODB*.
- **ODB**: representa una conexión con una base de datos orientada a objetos almacenada en un fichero. A través del objeto *ODB*, en nuestro ejemplo *odbAlumnos*, podemos realizar las cuatro clásicas operaciones CRUD (*Create, Read, Update and Delete*).

3. En tercer lugar, persistimos los tres objetos *Alumno* creados.

El método **store** de la clase *ODB* recibe como argumento un objeto, que será persistido en la base de datos. Corresponde con la C (*Create*) de las cuatro operaciones CRUD.

```
odbAlumnos.store(alu1);
odbAlumnos.store(alu2);
odbAlumnos.store(alu3);
```

Persistimos tres objetos en la base de datos *ALUMNOS.DB*, es decir, los guardamos.

4. En cuarto lugar, cerramos la conexión con la base de datos.

Una vez realizadas las operaciones deseadas, habrá que cerrar la conexión con la base de datos orientada a objetos a través del método *close* de la clase *ODB*.

```
odbAlumnos.close();
```

Si ejecutas varias veces el programa, se guardarán varias veces los mismos objetos, creando duplicados. Ten en cuenta que el método *open* crea la base de datos si no existe, pero si existe, la abre permitiendo añadir nuevos objetos sin eliminar los existentes.

Examinar base de datos desde NeoDatis

En este apartado, volverás a **abrir la interfaz de usuario de NeoDatis para examinar la base de datos** que acabas de crear.

1. Abre la aplicación NeoDatis.

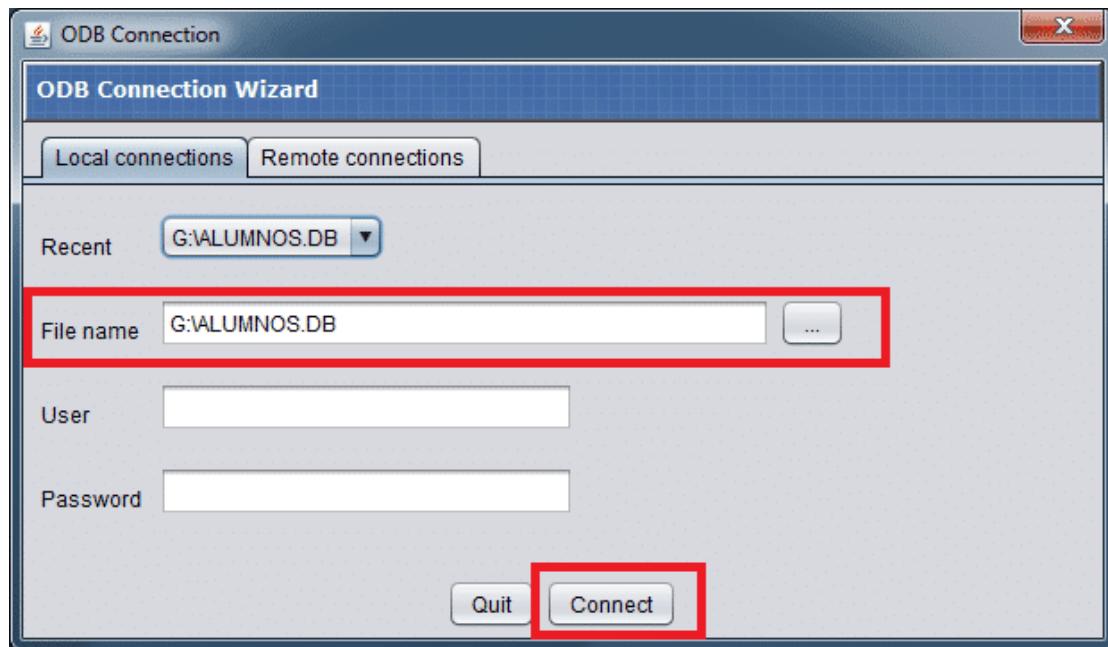
Accede a la carpeta donde descomprimiste NeoDatis y haz clic sobre el archivo ***odb-explorer.bat*** para abrir el entorno de usuario de *NeoDatis*.

2. Abre la base de datos ***ALUMNOS.DB***.

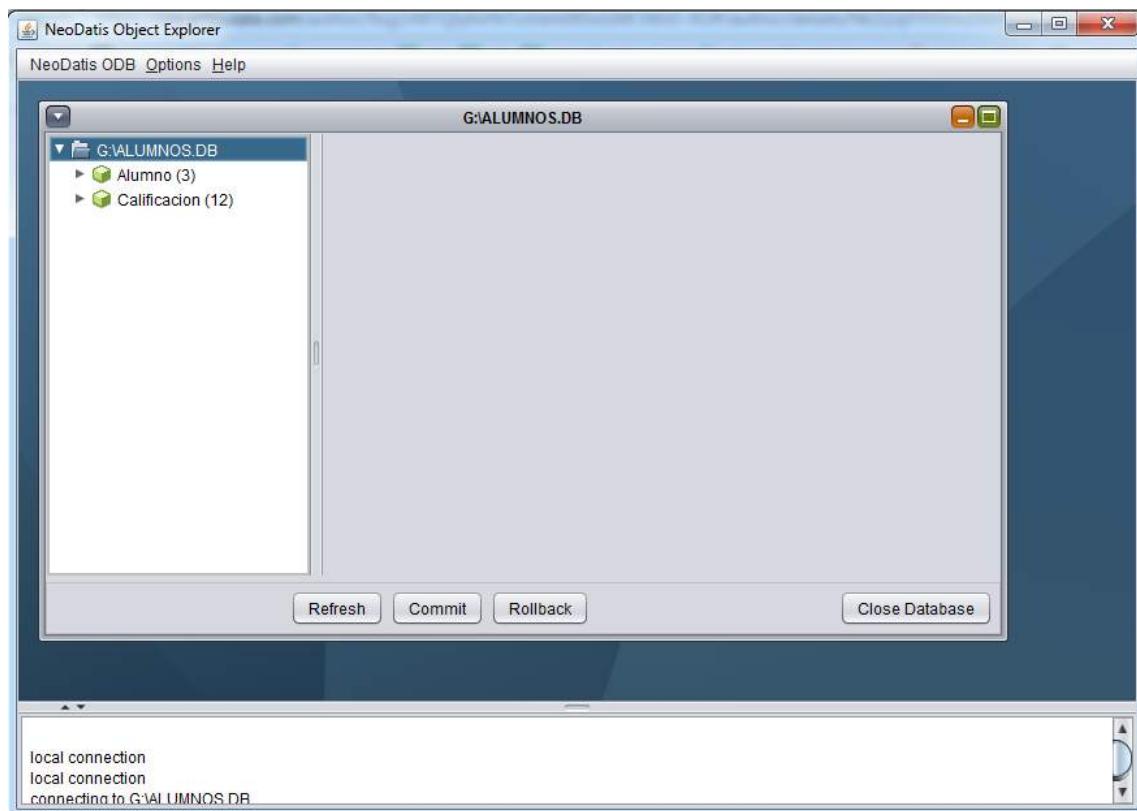
Para ello, selecciona en el menú **NeoDatis ODB / Open Database**.



Debes especificar la ruta y el nombre de la base de datos en el campo **File Name**. Puedes hacer clic en el botón de los tres puntos, situado a la derecha, para obtener el cuadro de diálogo *Abrir* que te permitirá seleccionar el archivo.



Finalmente, haz clic en el botón **Connect** y tendrás una vista de la base de datos **ALUMNOS.DB**. Éste será el aspecto de la vista de la base de datos:



Aspecto final de tu base de datos.

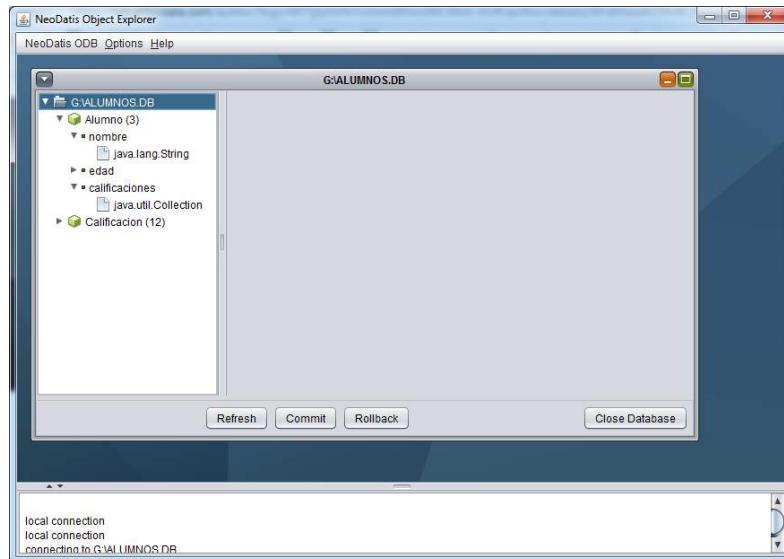
Distintas vistas para examinar la BD

La interfaz de usuario de NeoDatis cuenta con **varias vistas para examinar los objetos almacenados en la base de datos.**

¿Quieres descubrirlos?

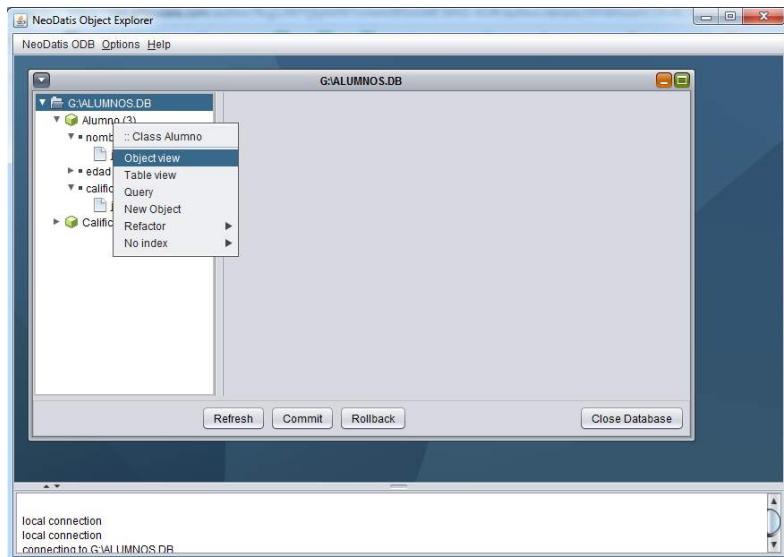
Lo que NeoDatis te está mostrando en un primer momento es la estructura de las clases a las que pertenecen los objetos almacenados en la base de datos.

Por la imagen, puedes comprobar que la base de datos tiene almacenados tres objetos *Alumno* y doce objetos *Calificacion*.



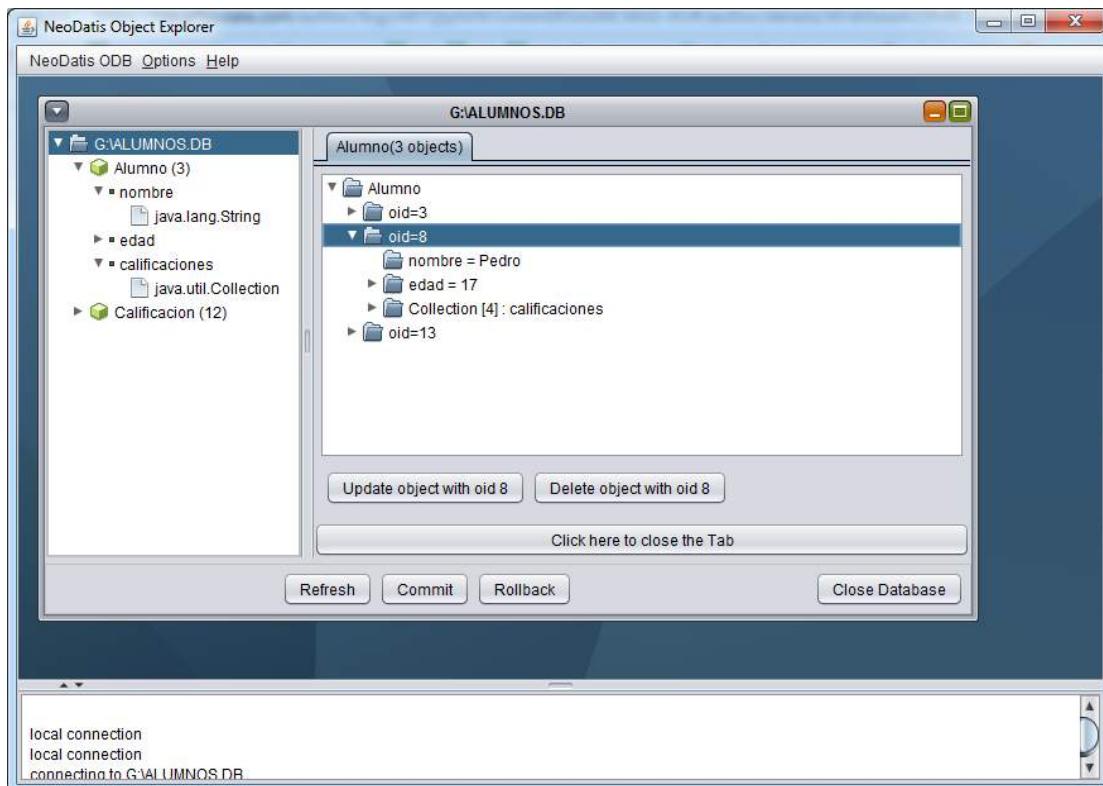
A la izquierda del nombre de la clase, hay un triángulo que te permitirá expandir o contraer los atributos que componen cada clase. Junto al atributo también aparece un triángulo, que te permitirá mostrar u ocultar el tipo de dato al que pertenece.

Puedes hacer clic derecho sobre el nombre de una de las clases para seleccionar una entre varias vistas.



Object View

Selecciona la vista **Object View**.



Object View.

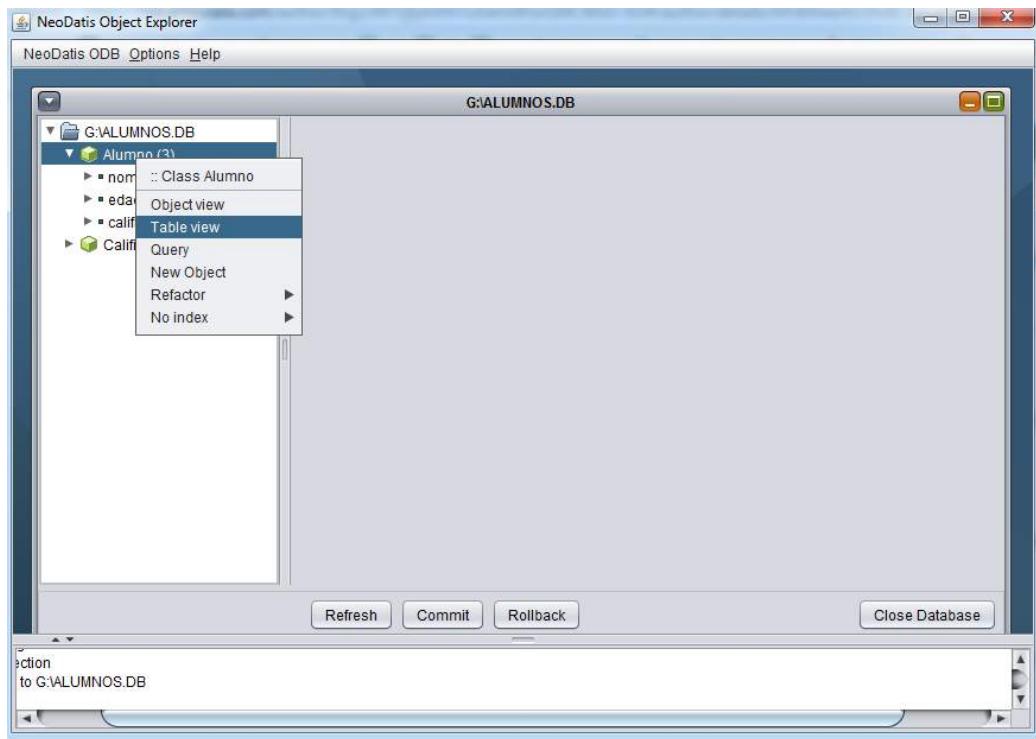
Puedes utilizar los botones *expandir* y *contraer* (representados por triángulos) para ver el objeto con más o menos detalle.

También puedes utilizar los botones *Update object* o *Delete object* para editar o eliminar el objeto seleccionado.

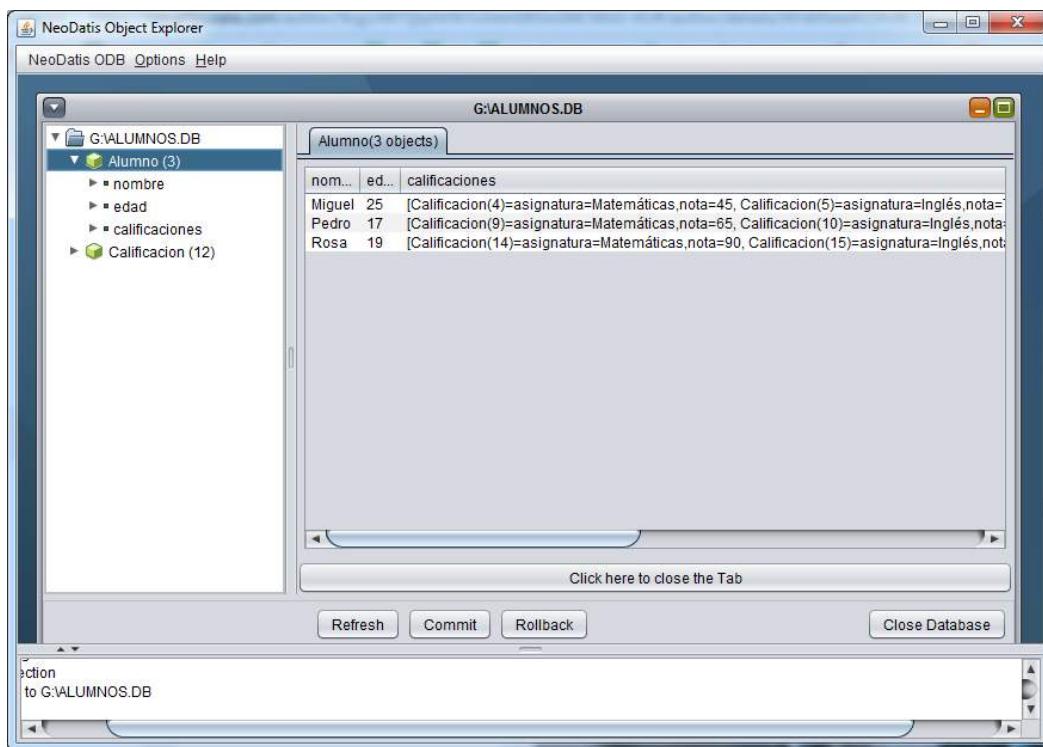
Cuando quieras cerrar la vista, sólo tienes que hacer clic en el botón *Click here to close the tab*.

Table View

Vuelve a hacer clic derecho sobre la clase *Alumno*, pero esta vez selecciona la opción *Table View*.

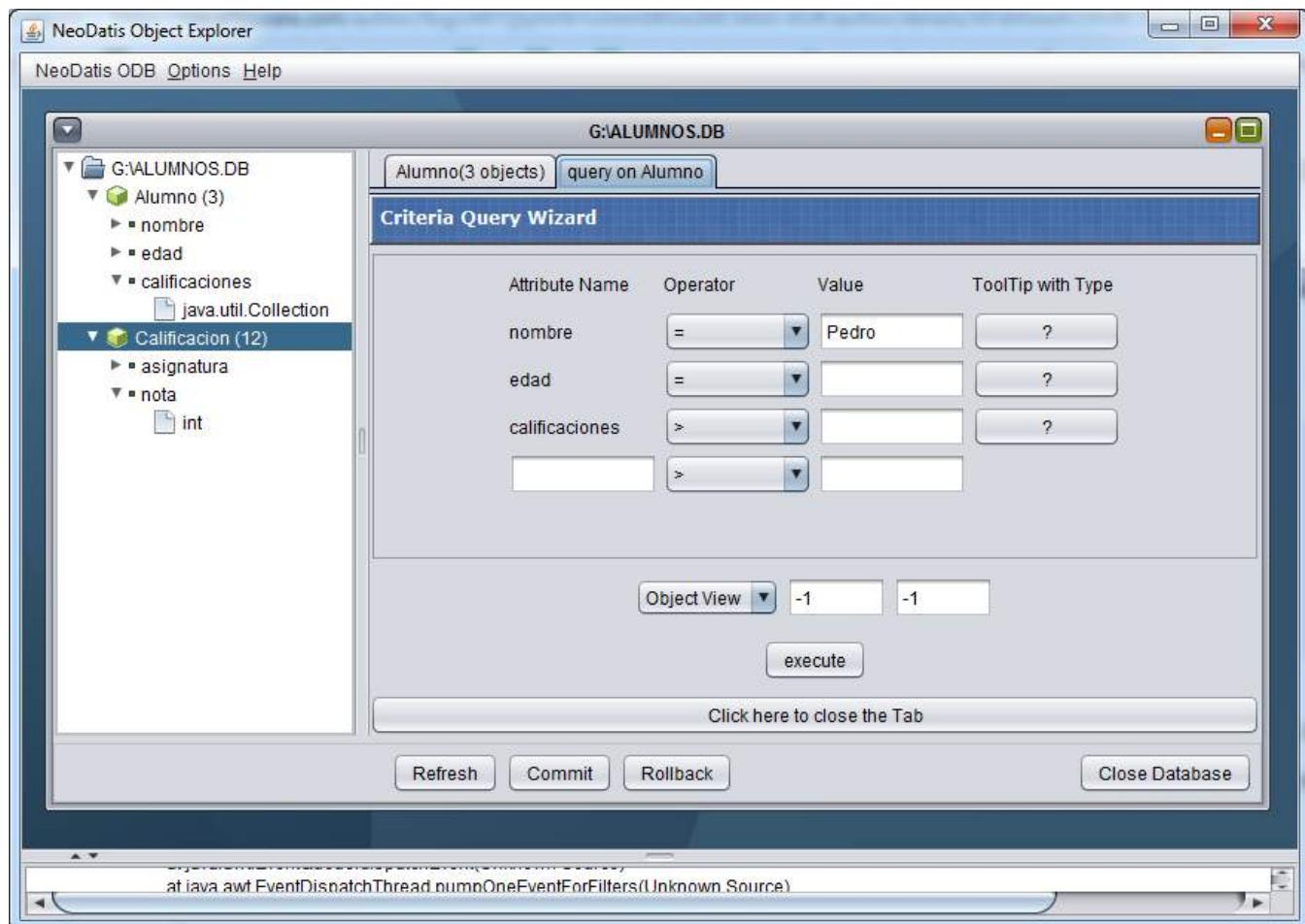


Ahora estás viendo los objetos organizados en una tabla con filas y columnas.



Query

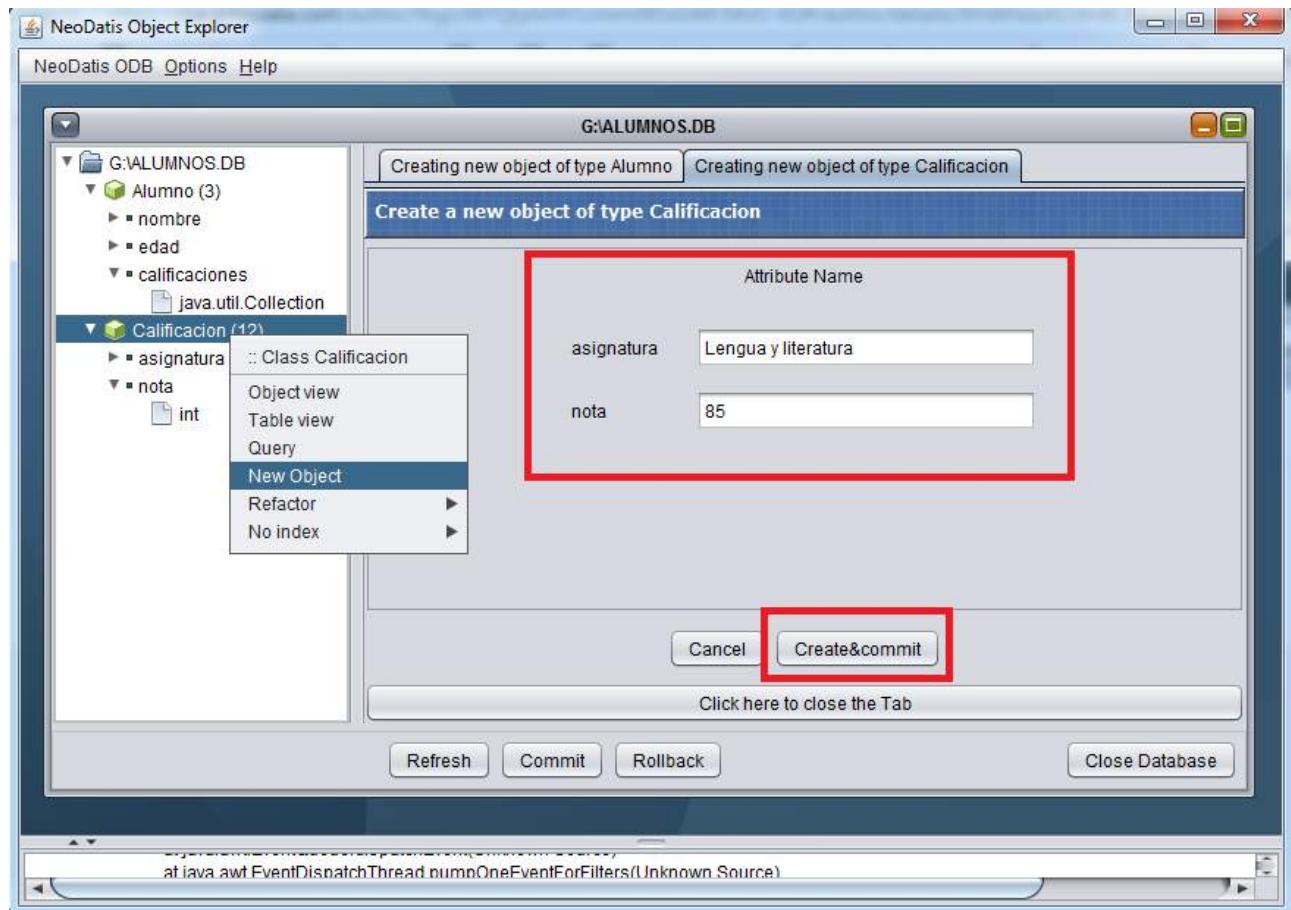
También puedes hacer clic derecho en el nombre de una clase y seleccionar la opción *Query* para localizar los objetos que cumplan un criterio de búsqueda. En el ejemplo de la imagen, queremos localizar al alumno con *nombre* = "Pedro".



Al hacer clic en el botón *Execute* obtendrás el resultado de la búsqueda.

New Object

Incluso es posible añadir nuevos objetos en la base de datos, haciendo clic derecho en el nombre de la clase y seleccionando la opción *New Object*.



Consultar BD Alumnos

Consultar base de datos de alumnos en Java

En este apartado crearemos un **programa Java que permita leer secuencialmente los objetos almacenados** en la base de datos *ALUMNOS.DB*.

Dentro del mismo proyecto, crea otra clase con método *main* con el nombre *RecuperarObjetos*.
Copia y pega el código siguiente; más tarde lo analizaremos detenidamente.

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

public class RecuperarObjetos {
    public static void main(String args[]) {

        ODB objAlumnos = ODBFactory.open("G:/ALUMNOS.DB");

        Objects<Alumno> alumnos=objAlumnos.getObjects(Alumno.class);

        Alumno alu;
        while (alumnos.hasNext()) {
            alu=alumnos.next();
            System.out.println(alu.getNombre() + " - " + alu.getEdad() + " "
años");
            for (Calificacion c : alu.getCalificaciones()) {
                System.out.println(" " + c);
            }
        }

        objAlumnos.close();
    }
}
```

En primer lugar, nos centraremos en esta línea:

```
Objects<Alumno> alumnos=objAlumnos.getObjects(Alumno.class);
```

Si lo que deseamos es recuperar todos los objetos de la clase *Alumno* almacenados en la base de datos, bastará con ejecutar el método ***getObjects()*** de la clase ***ODB***, pasando como argumento la clase cuyos objetos deseamos recuperar, en este caso *Alumno.class*. El método *getObjects()* devuelve un objeto de la clase genérica ***Objects*** que contiene la colección de objetos recuperados.

Los objetos de la clase *Objects* actúan como cursores que pueden recorrerse hacia adelante con el método *next()*. Para controlar el final de fichero utilizamos el método *hasNext()*, que devuelve *true*, si hay más objetos pendientes de lectura, y *false* en caso contrario.

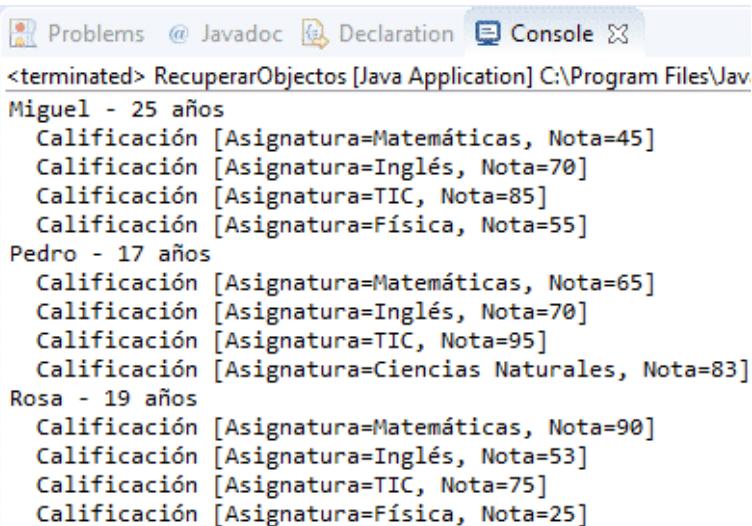
```
alu=alumnos.next();
```

Con esta línea, recuperamos el siguiente objeto *Alumno*. La variable *alu* será la referencia que nos dé acceso al objeto.

```
for (Calificacion c : alu.getCalificaciones()) {  
    System.out.println(" " + c);  
}
```

Sabemos que un objeto *Alumno* contiene objetos *Calificacion*, así que las recorremos con una estructura *for*.

Si has llegado a ejecutar el programa, el resultado habrá sido éste:



```
<terminated> RecuperarObjectos [Java Application] C:\Program Files\Java  
Miguel - 25 años  
    Calificación [Asignatura=Matemáticas, Nota=45]  
    Calificación [Asignatura=Inglés, Nota=70]  
    Calificación [Asignatura=TIC, Nota=85]  
    Calificación [Asignatura=Física, Nota=55]  
Pedro - 17 años  
    Calificación [Asignatura=Matemáticas, Nota=65]  
    Calificación [Asignatura=Inglés, Nota=70]  
    Calificación [Asignatura=TIC, Nota=95]  
    Calificación [Asignatura=Ciencias Naturales, Nota=83]  
Rosa - 19 años  
    Calificación [Asignatura=Matemáticas, Nota=90]  
    Calificación [Asignatura=Inglés, Nota=53]  
    Calificación [Asignatura=TIC, Nota=75]  
    Calificación [Asignatura=Física, Nota=25]
```

Resultado final.

Importar y exportar en Neodatis

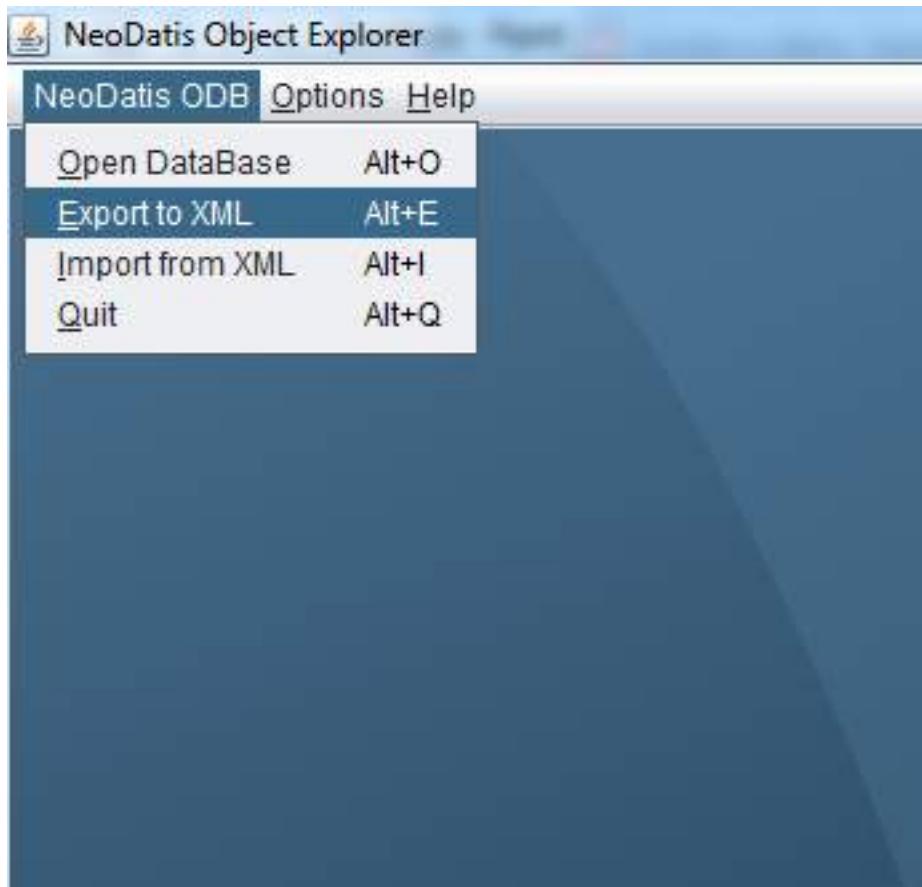
Exportar Object DB a XML

Dentro de la interfaz de usuario de NeoDatis es posible **exportar la base de datos a formato XML**.

De esta forma tendremos una copia de seguridad de los datos.

¿Quieres ponerlo en práctica?

Abre la interfaz de usuario de NeoDatis y selecciona en el menú *NeoDatis ODB / Export to XML*.

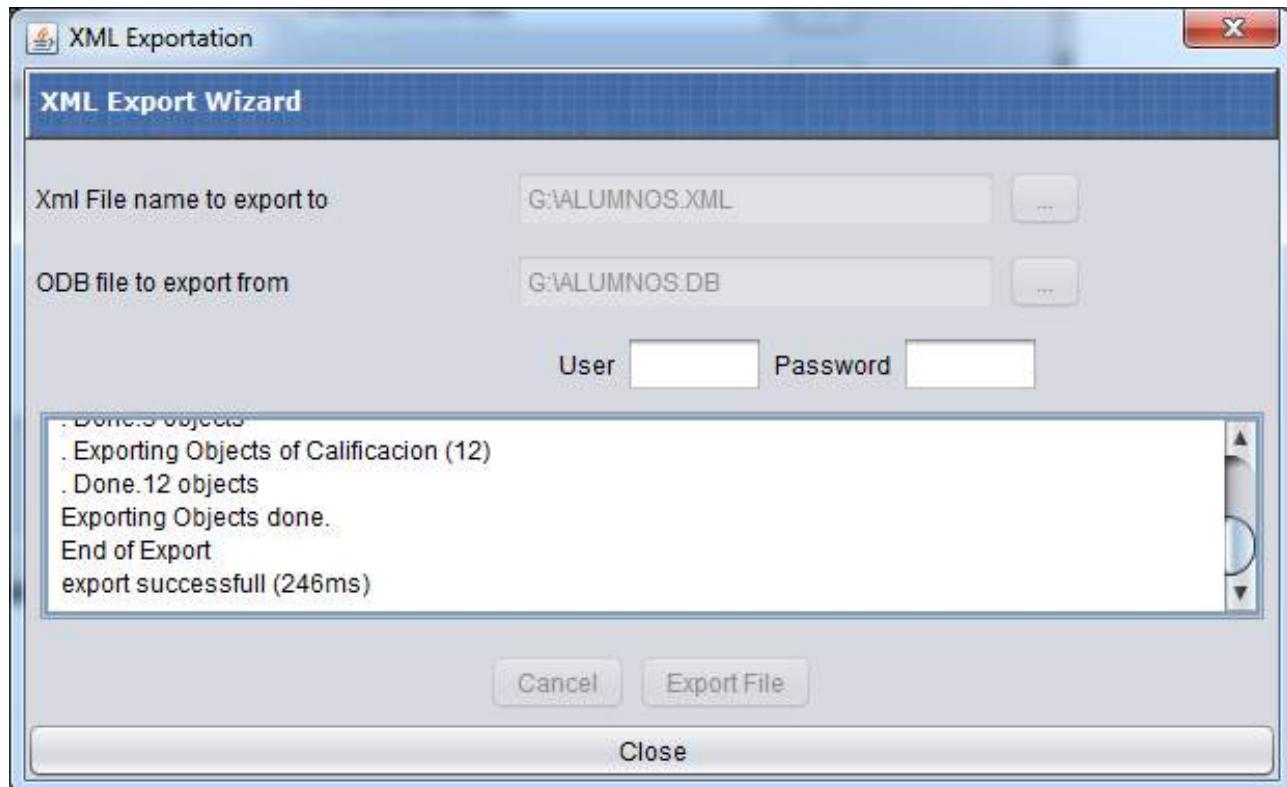


Dentro del cuadro de diálogo *XML Exportation*, debes llenar dos campos:

- ***XmL File name to export to:*** escribe aquí la ruta y nombre del futuro fichero XML de destino.
- ***ODB file to export from:*** escribe aquí la ruta y nombre del archivo que contiene la base de datos de origen.



Finaliza haciendo clic en el botón *Export File*. Si la exportación se ha efectuado con éxito, en la consola de resultados verás el mensaje final *export successfull*.



Ya puedes hacer clic en el botón *Close*. El archivo XML que has obtenido tendrá el siguiente formato:



```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <odb file-format-version="9" max-oid="18" export-date-time="1530814840879" name="ALUMNOS.DB">
  - <meta-model>
    - <class name="Alumno" id="1">
      <attribute name="nombre" id="1" type="java.lang.String"/>
      <attribute name="edad" id="2" type="int"/>
      <attribute name="calificaciones" id="3" type="java.util.Collection"/>
    </class>
    - <class name="Calificacion" id="2">
      <attribute name="asignatura" id="1" type="java.lang.String"/>
      <attribute name="nota" id="2" type="int"/>
    </class>
  </meta-model>
  - <objects>
    - <object class-id="1" oid="3">
      <attribute name="nombre" id="1" value="Miguel"/>
      <attribute name="edad" id="2" value="25"/>
      - <attribute name="calificaciones" id="3" type="collection">
        - <collection size="4" native-class-name="java.util.ArrayList">
          <element ref-oid="4"/>
          <element ref-oid="5"/>
          <element ref-oid="6"/>
          <element ref-oid="7"/>
        </collection>
      </attribute>
    </object>
    - <object class-id="1" oid="8">
      <attribute name="nombre" id="1" value="Pedro"/>
      <attribute name="edad" id="2" value="17"/>
      - <attribute name="calificaciones" id="3" type="collection">
        - <collection size="4" native-class-name="java.util.ArrayList">
```

Como puedes comprobar, el documento XML contiene, tanto la estructura de las clases, como los datos guardados en los objetos. Si se llegara a deteriorar el archivo *ALUMNOS.DB* podríamos volver a construirlo a partir del documento PDF, ejecutando la opción *Import from XML* en el menú de la interfaz de usuario de NeoDatis.

Ya hemos visto cómo se exporta un objeto DB a XML. Veamos ahora cómo se importa.

Importar XML para obtener Object DB

En este apartado realizarás la **importación de una copia de seguridad de una base de datos almacenada en formato XML**. La base de datos contendrá el inventario de un almacén de productos de alimentación.

Para este ejemplo, necesitas el archivo "almacen.xml" que puedes descargar de la versión online de esta lección.

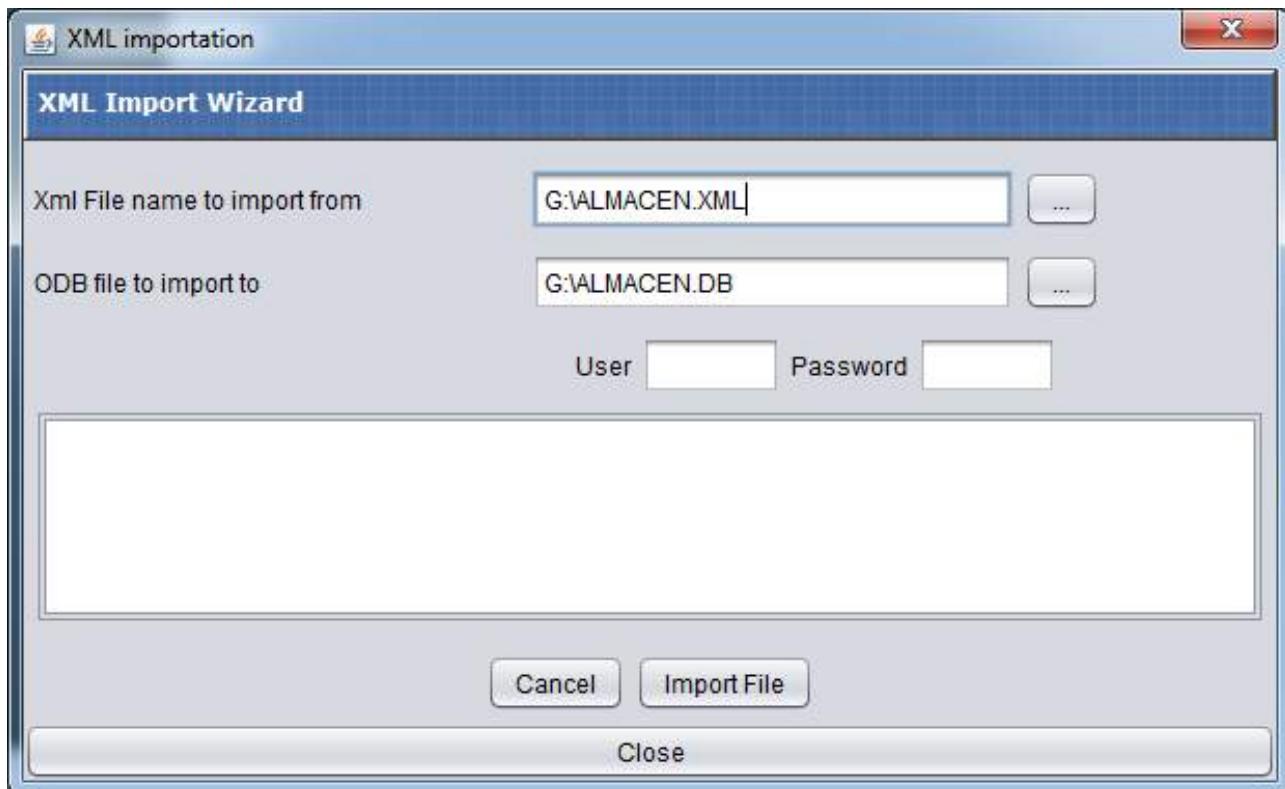
Si ya has realizado la descarga del archivo, pongámonos manos a la obra para importarlo y obtener la base de datos orientada a objetos.

Abre la interfaz de usuario de NeoDatis y selecciona en el menú **NeoDatis ODB / Import from XML**.



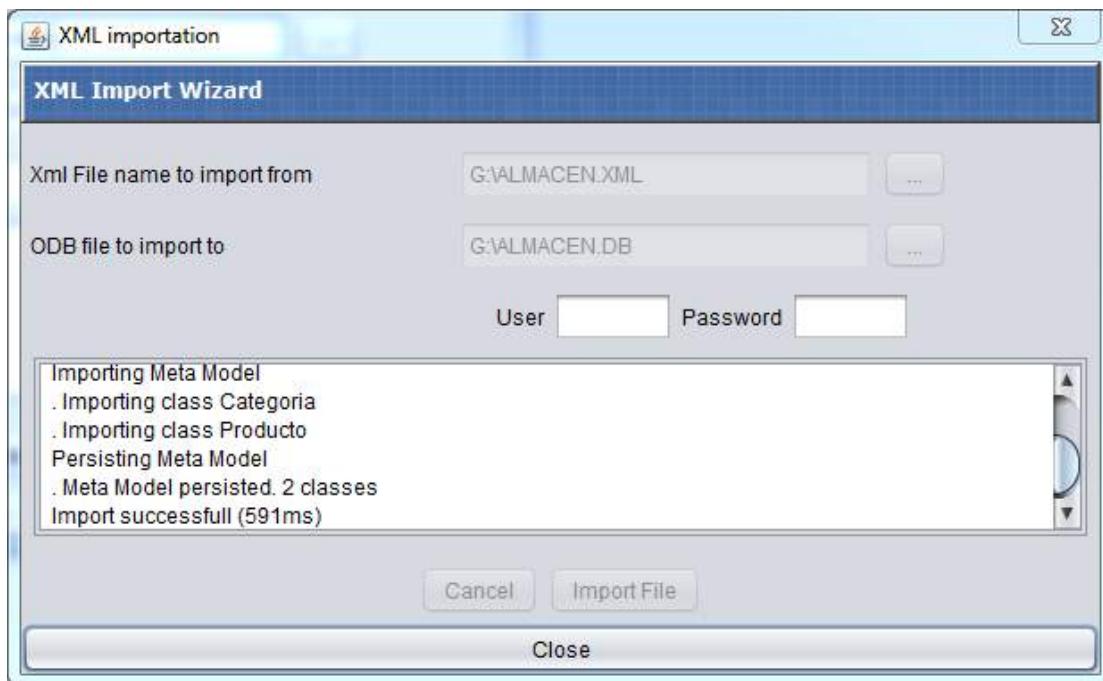
Dentro del cuadro de diálogo **XML importation** tendrás que llenar dos campos:

- **Xml File name to import from:** escribe aquí la ruta y nombre del archivo XML que has descargado. Puedes utilizar el botón de la derecha, con tres puntos, para seleccionar la carpeta y el archivo.
- **ODB file to import to:** escribe aquí la ruta y nombre de la nueva base de datos que será el destino de la importación.



Por último, haz clic en el botón **Import File**.

Si la importación se ha efectuado con éxito, en la consola de resultados verás el mensaje final *Import successfull*. Ya puedes hacer clic en el botón **Close** para cerrar el cuadro de diálogo.

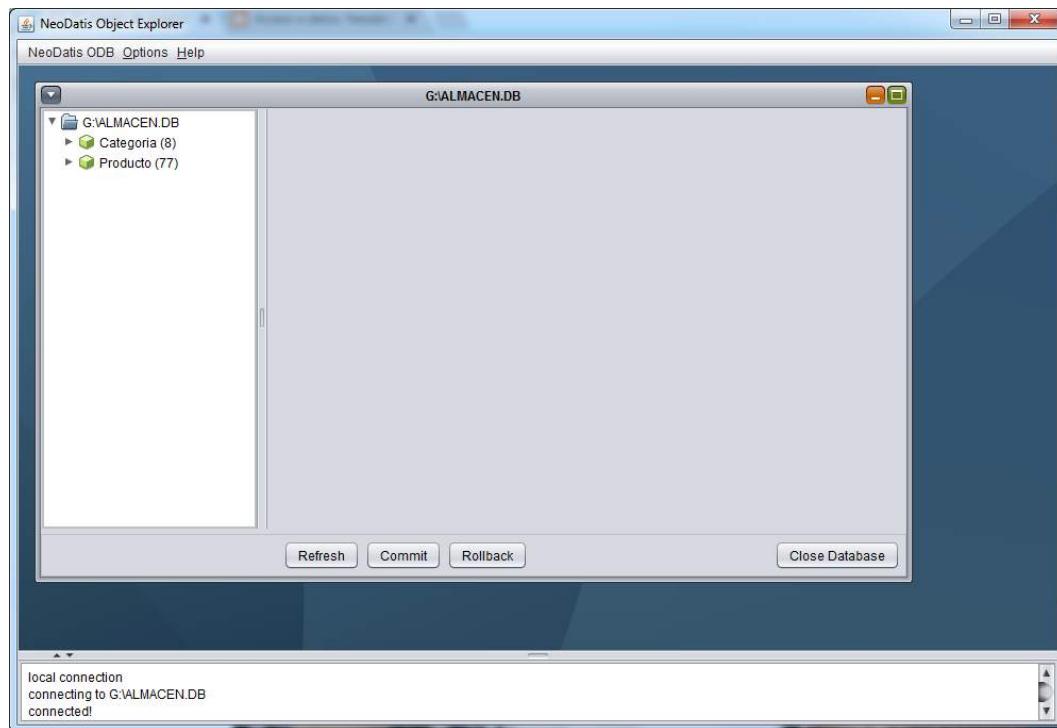


Para comprobar que la importación ha sido un éxito, puedes abrir la base de datos, explorar y familiarizarte con el contenido.

Selecciona en el menú **NeoDatis ODB / Open Database**. Escribe la ruta y nombre del nuevo archivo de base de datos en el campo **File name**.



Para terminar, haz clic en el botón **Connect**.



La interfaz de usuario de NeoDatis te está indicando que en la base de datos hay almacenados ocho objetos de la clase *Categoría* y, setenta y siete objetos de la clase *Producto*. Existe una asociación entre ellos, ya que cada objeto *Categoría* está compuesto por los atributos *nombre* y *productos* (colección de objetos *Producto*).

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Dentro de la carpeta de la aplicación NeoDatis se encuentra el archivo ***neodatis-odb-1.9.30.689.jar*** que contiene las librerías de clases necesarias para formar parte de la capa de persistencia de nuestras aplicaciones Java. Haciendo uso de dichas clases, grabar y recuperar objetos es una tarea muy sencilla.
- Desde la interfaz de usuario de NeoDatis podemos **examinar los objetos almacenados en las bases de datos** que hemos creado previamente desde nuestra aplicación Java.
- La interfaz de usuario de NeoDatis nos permite crear **copias de seguridad de la base de datos en formato XML** por medio de la opción de menú ***Export to XML***. Si la base de datos original llegara a deteriorarse, podríamos restaurarla importando los datos desde la copia XML a través de la opción de menú ***Import from XML***.

6.4. Lenguaje de consultas para objetos



Índice

Objetivos	3
El lenguaje OQL.....	4
Introducción.....	4
Ejemplos de consultas OQL	4
1. Ejemplo 1	4
2. Ejemplo 2	5
3. Ejemplo 3	5
4. Ejemplo 4	5
OQL en Neodatis: Programación JAVA.....	6
Consultas con la librería NeoDatis	6
Ejemplos resueltos.....	9
Ordenar los resultados.....	10
Consultas avanzadas	11
La clase And	11
La clase Or	11
Combinar And y Or	12
Despedida	14
Resumen.....	14

Objetivos

En esta lección perseguimos los siguientes objetivos:

- Comprender el significado del estándar OQL para las consultas en bases de datos orientadas a objetos.
- Desarrollar programas Java que ejecuten consultas sobre bases de datos orientadas a objetos.

El lenguaje OQL

Introducción

OQL (*Object Query Language*) es un estándar para creación de consultas en bases de datos orientadas a objetos que nace a partir de SQL (*Structured Query Language*).

Como estándar, fue desarrollado por la organización Object Data Management Group.

OQL todavía está en vías de desarrollo, ya que debido a su complejidad ningún creador de software lo ha implementado completamente.

No hay que olvidar que se trata de un estándar, de modo que cada fabricante desarrolla OQL a su manera cumpliendo las normas que dicta el estándar.

La idea consiste en aprovechar la sintaxis de cada una de las sentencias del lenguaje SQL estándar para crear una sintaxis lo más parecida posible, pero trasladada al trabajo con objetos.

Ejemplos de consultas OQL

Para que comprendas mejor la sintaxis de las consultas OQL, vamos a ver algunos **ejemplos comparados con el estándar SQL**.

1. Ejemplo 1

SQL estándar

```
SELECT * FROM Producto  
WHERE nombre="Arenque ahumado";
```

OQL

```
select p from p in Producto  
where p.nombre="Arenque ahumado";
```

La variable p contendrá la referencia de cada objeto leído de la clase Producto. En el ejemplo, recuperaremos un objeto de la clase Producto completo, en concreto, un producto cuyo nombre sea "Arenque ahumado".

2. Ejemplo 2

SQL estándar

```
SELECT precio FROM Producto  
WHERE nombre="Arenque ahumado";
```

OQL

```
select p.precio from p in Producto  
where p.nombre="Arenque ahumado";
```

Este ejemplo es similar al anterior, pero en lugar de recuperar el objeto *Producto* completo, sólo recuperamos un dato elemental, el valor del atributo *precio*.

3. Ejemplo 3

SQL estándar

```
SELECT nombre, precio, stock  
FROM Producto  
WHERE nombre="Arenque ahumado";
```

OQL

```
select struct(nom: p.nombre, pre: p.precio, st: p.stock)  
from p in Producto  
where p.nombre ="Arenque ahumado";
```

La variable *p* sigue conteniendo la referencia al objeto *Producto*, pero ahora no deseamos recuperar el objeto *Producto* completo, sino un subconjunto (estructura) formado por los atributos *nombre*, *precio* y *stock*. En las variables *nom*, *pre* y *st* se guardarán los valores de los atributos *nombre*, *precio* y *stock*.

4. Ejemplo 4

SQL estándar

```
SELECT * FROM Producto  
WHERE precio > 50  
ORDER BY precio;
```

OQL

```
select p from p in Producto  
where p.precio > 50  
order by p.precio;
```

En este caso, recuperamos los objetos de la clase *Producto* cuyo precio sea mayor a 50€.

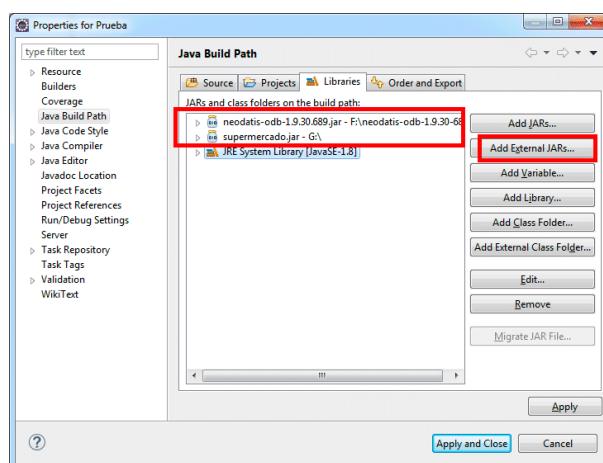
OQL en Neodatis: Programación JAVA

Preparación del proyecto en Eclipse

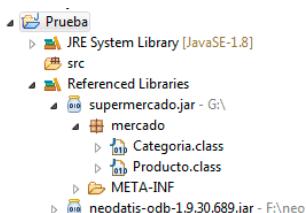
Para que puedas poner en práctica los ejemplos que te mostraremos en los próximos apartados, primero tendrás que preparar el proyecto estándar java en Eclipse con los recursos necesarios.

¡Vamos manos a la obra!

- 1. Crea un proyecto estándar Java** con el nombre que deseas (*File / New / Java Project*).
- 2. Trabajará con la base de datos *ALMACEN.DB* que importaste en la lección anterior**, pero necesitarás disponer de las clases *Categoría* y *Producto* para poder recoger los resultados de las consultas OQL que ejecutes. Para facilitarte la labor, hemos empaquetado dichas clases en una librería denominada *supermercado.jar* que podrás descargar en la versión online de esta lección. Una vez descargado, renombra el archivo para que mantenga el nombre de *supermercado.jar*.
- 3. Tendrás que acceder a las propiedades del proyecto**, concretamente al apartado "Java Build Path" para **importar** la librería *supermercado.jar* y también la **librería de clases de NeoDatis**.



- 4. Despliega la librería *supermercado.jar* para comprobar como está organizada.**



Puedes comprobar que contiene las clases *Categoría* y *Producto* empaquetadas dentro del paquete *mercado*.

Ya tienes preparado el proyecto que te servirá de base para poner en práctica los ejemplos de los siguientes apartados.

Consultas con la librería NeoDatis

En este apartado tendrás la oportunidad de **ejecutar consultas OQL en tus programas Java con ayuda de la librería de clases de NeoDatis**.

En el apartado anterior viste el **formato estándar de OQL**. Ahora comprobarás que NeoDatis tiene su forma particular de implementarlo, tal como ocurre con cada gestor de bases de datos orientadas a objetos.

Cuando no deseamos obtener todos los objetos de una determinada clase almacenados en la base de datos, sino sólo aquellos que cumplan algún criterio o condición *where*, necesitamos definir un objeto de la clase **CriteriaQuery** que se construye con dos argumentos: el primero será la clase a la que pertenecen los objetos que deseamos obtener, y el segundo la expresión condicional que determinará los objetos que se van a recuperar. Veamos un ejemplo:

```
CriteriaQuery consulta = new CriteriaQuery(Categoría.class,  
Where.equal("nombre", "Pescado/Marisco"));
```

Nuestro objeto *consulta* de la clase *CriteriaQuery* representa una condición de búsqueda sobre los objetos de la clase *Categoría*, cuya condición es que el atributo *nombre* sea igual a *Pescado/Marisco*. Este objeto puede ser pasado como argumento al método *getObject()* de la clase ODB.

```
Objects<Categoría> categorías = objCategoría.getObject(consulta);
```

El ejemplo completo quedaría así:

```
import org.neodatis.odb.ODB;  
import org.neodatis.odb.ODBFactory;  
import org.neodatis.odb.Objects;  
import org.neodatis.odb.core.query.criteria.Where;  
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;  
  
public class Consultas {  
    public static void main(String[] args) {  
        ODB objCategoría = ODBFactory.open("G:/ALMACEN.DB");  
  
        CriteriaQuery consulta = new CriteriaQuery(Categoría.class, Where.equal("nombre",  
"Pescado/Marisco"));  
        Objects<Categoría> categorías = objCategoría.getObject(consulta);  
  
        Categoría cat;  
        while (categorías.hasNext()) {  
            cat=categorías.next();  
            System.out.println(cat.getNombre());  
            for (Producto p : cat.getProductos()) {  
                System.out.println(" " + p.getNombre());  
            }  
        }  
        objCategoría.close();  
    }  
}
```

Para cada objeto *Categoría* obtenido, recorremos sus productos a partir del método *getProductos()*.

También podemos dividir el proceso en dos procesos más pequeños, creando un objeto **ICriterion** en el que se especifica el criterio de búsqueda. Luego, pasamos ese objeto como segundo argumento de nuestro *CriteriaQuery*. El proceso quedaría así:

```
ICriterion criterio = Where.equal("nombre", "Pescado/Marisco");  
CriteriaQuery consulta = new CriteriaQuery(Categoría.class, criterio);  
Objects<Categoría> categorías = objCategoría.getObjects(consulta);
```

De esta forma, quedará más claro cuando los criterios de búsqueda comiencen a complicarse de verdad.

Presta atención ahora a la expresión siguiente:

```
Where.equal("nombre", "Pescado/Marisco")
```

Where es una clase con métodos estáticos que sirven para definir expresiones condicionales, devolviendo un objeto de tipo **ICriterion**. Podemos utilizar uno de los siguientes métodos:

- **Where.equals("atributo", "valor");**
El atributo debe ser igual al valor.
- **Where.gt("atributo", "valor");**
El atributo debe ser mayor que el valor.
- **Where.ge("atributo", "valor");**
El atributo debe ser mayor o igual que el valor.
- **Where.lt("atributo", "valor");**
El atributo debe ser menor que el valor.
- **Where.le("atributo", "valor");**
El atributo debe ser menor o igual que el valor.
- **Where.not("atributo", "valor");**
El atributo debe ser distinto que el valor.
- **Where.isNull("atributo");**
El atributo tiene que ser nulo.
- **Where.isNotNull("atributo");**
El atributo no tiene que ser nulo.
- **Where.like("atributo", "patrón de búsqueda");**
Permite especificar un patrón de búsqueda. Ejemplo:
ICriterion criterio = Where.like("nombre", "Queso%");
Productos cuyo nombre comienza con la palabra Queso seguida de cualquier combinación de caracteres.

En los siguientes apartados realizaremos más programas de ejemplo.

Ejemplos resueltos

1. Productos que cuestan más de 70 euros

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        ICriterion criterio = Where.gt("precio", 70);
        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

2. Productos cuyo nombre comienza con la letra A

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        ICriterion criterio = Where.like("nombre","A%");
        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

Ordenar los resultados

La clase ***CriteriaQuery*** cuenta con dos métodos que permiten **obtener los resultados de la consulta ordenados de forma ascendente o descendente**.

Se trata de los métodos ***orderByAsc()*** y ***orderByDesc()***. El siguiente ejemplo muestra los productos que cuestan más de 50 euros y los ordena descendente, según el valor del atributo *precio*.

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        ICriterion criterio = Where.gt("precio", 50);
        CriteriaQuery consulta = new CriteriaQuery(Producto.class,
criterio);
        consulta.orderByDesc("precio");
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

Consultas avanzadas

La clase And

Ya sabemos ejecutar consultas condicionales. Pero, ¿qué pasa cuando queremos expresar criterios más complejos? ¿Y si quiero que se cumplan dos o más condiciones?

La respuesta está en la clase *And*.

Cada objeto de la clase *And* representa una expresión condicional compuesta por una colección de criterios (objetos *ICriterion*), que deberán cumplirse todos para que dicha expresión condicional sea evaluada como verdadera. Una vez configurado correctamente, un objeto de la clase *And* puede ser pasado como argumento al constructor de la clase *CriteriaQuery*.

Veamos un ejemplo:

```
And criterio = new And();
criterio.add(Where.ge("precio", 10));
criterio.add(Where.le("precio", 50));
CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
```

Obtenemos los productos cuyo *precio* esté comprendido entre 10 y 50 euros, es decir, precio mayor o igual a 10, y precio menor o igual a 50.

En el ejemplo puedes ver más claramente que se trata de una colección de objetos *ICriterion*.

Una vez construido el objeto *And*, podemos utilizar el método *add()* para añadir tantos objetos *ICriterion* (es decir, condiciones) como sea necesario.

La clase Or

¿Y si deseamos establecer un conjunto de criterios, de manera que sólo deba cumplirse uno de ellos para cada objeto? En ese caso, **utilizaremos la clase *Or***.

Por ejemplo: queremos mostrar los productos cuyo nombre empiece por A, Q o S. Quedaría resuelto así:

```
Or criterio = new Or();
criterio.add(Where.like("nombre", "A%"));
criterio.add(Where.like("nombre", "Q%"));
criterio.add(Where.like("nombre", "S%"));
CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);
```

La clase *Or* funciona exactamente igual que la clase *And*; la diferencia está en que para que la expresión condicional sea verdadera, basta con que se cumpla cualquiera de las condiciones de la colección.

Y, ahora, el programa completo:

```

import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.Or;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        //And criterio = new And();
        //criterio.add(Where.ge("precio", 10));
        //criterio.add(Where.le("precio", 50));

        Or criterio = new Or();
        criterio.add(Where.like("nombre", "A%"));
        criterio.add(Where.like("nombre", "Q%"));
        criterio.add(Where.like("nombre", "S%"));
        CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio);

        consulta.orderByAsc("nombre");
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}

```

Combinar And y Or

Los objetos de las clases *And* y *Or* no dejan de ser objetos *ICriterion*, pero más complejos. Realmente, *ICriterion* es una interfaz y existen varias clases que implementan dicha interfaz y que, por lo tanto, también son clases *ICriterion*.

A un objeto de la clase *And* podemos añadirle un objeto *Or*, y viceversa. De este modo, podemos construir criterios mucho más complejos.

Veamos un ejemplo:

```

Or criterio1 = new Or();
criterio1.add(Where.like("nombre", "A%"));
criterio1.add(Where.like("nombre", "Q%"));
criterio1.add(Where.like("nombre", "S%"));

And criterio2 = new And();
criterio2.add(Where.ge("precio", 10));
criterio2.add(Where.le("precio", 50));
criterio2.add(criterio1);

CriteriaQuery consulta = new CriteriaQuery(Producto.class, criterio2);

```

En el ejemplo, estamos construyendo el objeto *CriteriaQuery* pasando como segundo argumento la variable *criterio2*. Dicha variable expresa una condición de tipo *And*, configurada de tal modo que deben cumplirse tres condiciones: que el *precio* sea mayor o igual a 10, que el *precio* sea menor o igual a 50 y que el *nombre* comience por A, Q o S. La última condición se ha configurado por medio de un objeto *Or*. Y, ahora, el programa completo:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.criteria.And;
import org.neodatis.odb.core.query.criteria.Or;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class Consultas {
    public static void main(String[] args) {
        ODB objProductos = ODBFactory.open("G:/ALMACEN.DB");

        Or criterio1 = new Or();
        criterio1.add(Where.like("nombre", "A%"));
        criterio1.add(Where.like("nombre", "Q%"));
        criterio1.add(Where.like("nombre", "S%"));

        And criterio2 = new And();
        criterio2.add(Where.ge("precio", 10));
        criterio2.add(Where.le("precio", 50));

        criterio2.add(criterio1);

        CriteriaQuery consulta = new CriteriaQuery(Producto.class,
criterio2);

        consulta.orderByAsc("precio");
        Objects<Producto> productos = objProductos.getObjects(consulta);
        System.out.println("Hay " + productos.size() + " productos que
cumplen el criterio");
        Producto pro;
        while (productos.hasNext()) {
            pro=productos.next();
            System.out.println(pro.getNombre() + " - " + pro.getPrecio());
        }

        objProductos.close();
    }
}
```

Despedida

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- **OQL** (*Object Query Language*) es un estándar para creación de consultas en bases de datos orientadas a objetos que nace a partir de SQL (*Structured Query Language*).
- OQL todavía **está en vías de desarrollo**, ya que debido a su complejidad ningún creador de software lo ha implementado completamente.
- **NeoDatis** tiene su forma particular de implementar OQL. La clase principal relacionada con la ejecución de consultas OQL es ***CriteriaQuery***, que se construye con dos argumentos: el primero será la clase a la que pertenecen los objetos que deseamos obtener, y el segundo la expresión condicional que determinará los objetos que se van a recuperar.