ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES

MULTIPLATAFORMA

Introducción al manejo de ficheros

Definición y tipos de ficheros

Definimos un fichero como la sucesión de bits que finalmente son almacenados en un determinado dispositivo. Está compuesto por nombre (identifica al fichero) y extensión (indica la tipología del archivo). Podemos organizar los ficheros en 2 grandes familias:

- **De texto (ASCII):** en su interior encontraremos líneas de texto organizadas en código ASCII.
- **Binarios:** aquellos ficheros cuya información está representada en código binario.

La clase java.io.File

Nos será de gran ayuda para **crear, modificar y eliminar ficheros**, además de que nos proporcionará información adicional de los mismos. Puedes observar como se **instancia un objeto de clase *File***:

```
File fichero = new File (pathname: "/carpeta/archivo");
```

Una vez instanciado el objeto podremos también **crear dicho fichero**, para ello usaremos la siguiente línea:

```
fichero.createNewFile();
```

Por último, otra acción básica que deberemos de aprender para el uso y gestión de ficheros en Java es la **eliminación de ficheros** y directorios, usaremos la siguiente línea:

```
fichero.delete();
```

Otros métodos de utilidad

Una vez realizado el aprendizaje de la instanciación, creación y borrado de un fichero, veamos en código sobre cómo **mover un fichero**:

```
package accesoADatosMoverFichero;

import java.io.File;

public class AccesoADatosMoverFichero {

	public static void main(String[] args) {

		try{

		    File fileOrigen = new File("C:\\temp\\pruebas1.text");
		    File fileDestino = new File("C:\\temp\\pruebas\\
		                                pruebas2.text");

		    if(fileOrigen.renameTo(fileDestino)){
		        System.out.println("El fichero se movió correctamente!");
		    }else{
		        System.out.println("El fichero no pudo moverse");
		    }
		}catch(Exception e){
		    e.printStackTrace();
		}
	}
}
```

Para poder mover un fichero, lo primero de lo que hay que asegurarse, es que éste debe de existir. Por otra parte, al cargar la ruta del fichero origen “fileOrigen” y ejecutar el método renameTo(), debemos indicar al método, el fichero destino “fileDestino”. Hay que tener en cuenta que la **ruta destino** debe de ser una **ruta operativa y existente**.

Métodos más usados de la clase File:

Método	Descripción
createNewFile()	Genera el fichero indicado
delete()	Borra el fichero
mkdirs()	Crea directorio indicado
getName()	Devuelve String con nombre del fichero
getPath(), getAbsolutePath()	Devuelve la ruta relativa y absoluta
getParent()	Devuelve el directorio superior
renameTo()	Acepta como parámetro un nuevo File, el cual será la nueva ruta del fichero
exists()	Comprueba si existe el fichero
canWrite(), canRead()	Comprueba si puede ser escrito o leído
listFiles()	Devuelve un array con los ficheros del directorio indicado
lastModified()	Devuelve últimas modificaciones

Formas de acceder a un fichero

Cuando nos disponemos a acceder al contenido de un fichero es importante tener claro el criterio de acceso que vamos a usar. Disponemos de dos:

- Acceso **secuencial**.
- Acceso **aleatorio o directo**.

Previamente, tendremos que realizar un estudio del aplicativo a desarrollar para saber qué modo de acceso nos conviene. En el **modo secuencial**, como su nombre indica, la información de nuestro fichero es una secuencia de caracteres o bytes, de forma que, para acceder a un determinado byte o carácter del fichero, deberíamos de haber pasado previamente por todos los anteriores.

Si lo que deseamos es **acceder a un registro o posición determinada** de nuestro fichero, entonces nuestro método a escoger será el **modo aleatorio.**

La diferencia más notable entre estos dos tipos de acceso es la siguiente: mientras los ficheros de **acceso secuencial** deberán ser recorridos byte a byte o carácter a carácter con el tiempo, procesado de la información, y uso de recursos que esto conllevaría, mientras que, en los **ficheros de acceso aleatorio o directo**, se establecerá un **puntero en bytes**, el cual indicará la posición exacta donde vamos a realizar la lectura y/o escritura, y al que se podrá **acceder directamente**.

Tabla orientativa sobre algunas de las **clases que podremos utilizar (métodos de acceso)**:

Acceso basado en caracteres		Acceso basado en Bytes	
Entrada	Salida	Entrada	Salida
`FileReader`	`FileWriter`	`FileInputStream`	`FileOutputStream`
		`RandomAccessFile`	`RandomAccessFile`

Operaciones de gestión de ficheros. Clases asociadas

Clases de uso de acceso secuencial

Dentro de la familia del acceso secuencial, debemos tener en cuenta la posibilidad de trabajar con bytes o caracteres según nuestra conveniencia. Para ello vamos a exponer algunas de sus diferencias.

Acceso con bytes

Nos facilitan la lectura o escritura de un fichero como un **"stream" de bytes**. Para ello nos apoyaremos en la clase **FileInputStream (lectura** entrada de datos) y **FileOutputStream (escritura** salida de datos):

• **FileInputStream (lectura):** a continuación, se muestra un ejemplo de uso de esta clase:

```
InputStream entrada = new FileInputStream("prueba.txt");
```

Para empezar, instanciaremos el objeto de nuestra clase e indicaremos la ruta del fichero a cargar.

Con el método **read()** estaremos accediendo al **primer byte** del fichero. Éste devuelve un número entero, por lo tanto lo **asignamos a una variable entera**.

```
int data = entrada.read();
```

Una vez obtenido el primer byte, ya podremos trabajar con él. Con este método accedemos al primer Byte del fichero "en bruto" y nos servirá para **leer imágenes o archivos binarios**. Finalmente, cerramos el fichero y liberamos los recursos del sistema que estaban haciendo uso del "stream" de bytes.

A modo aclaración puntualizar que en estos ejemplos se ha suprimido el control de excepciones para mayor limpieza de código. Más adelante veremos cómo manejarlas.

FileOutputStream (escritura)

```
package uso_fileoutputstream;

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

public class CrearFicheroBytes {

	public static void main(String[] args) {
		String path = "salida.txt";
		String cadena = "Esto es una prueba de escritura";
		// pasamos a array de bytes la cadena de texto:
		byte[] arrayBytes = cadena.getBytes();

		try {
			// creamos el fichero en el path:
			FileOutputStream output = new FileOutputStream(path);
			// escribimos en el fichero:
			output.write(arrayBytes);
			output.close();
			System.out.println("Fichero escrito correctamente");
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

Acceso con caracteres

El modo de acceso secuencial con caracteres, hace posible la lectura y escritura de un fichero a través de un "stream" de caracteres. Mientras que las clases vistas previamente leen y escriben por medio de Bytes, las que veremos a continuación lo hacen **manipulando directamente caracteres**. Estas son: FileReader y FileWritter.

• **FileReader (lectura):** básicamente esta clase nos facilitará la lectura de los ficheros de caracteres. Pasamos a ver su utilización:

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class LecturaStreamDeCaracteres {
	public static void main(String[] args) {
		try {
			Reader lector = new FileReader("prueba2.txt");
			int data = lector.read();
			System.out.println((char)data);
			lector.close();
		} catch (FileNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

Como podemos observar, la lógica es muy parecida a la clase vista con anterioridad, FileInputStream. No obstante, ésta la usábamos para obtener los bytes “en bruto” directamente, mientras que la clase FileReader fue escrita con el propósito de leer **flujos** o “streams” **de caracteres**. Para ello, como podemos ver en el ejemplo, justo antes de mostrar el resultado por pantalla, realizamos un **casting (char)** para poder **visualizar** ese primer **carácter** de nuestro fichero “prueba2.txt”

• **FileWritter (escritura)**: con la siguiente clase seremos capaces de **escribir ficheros de caracteres**:

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

public class EscrituraStreamDeCaracteres {

	public static void main(String[] args) {
		try {
			// Se instancia clase de escritura y se indica ubicación de archivo
			// Si no existiera el archivo se crearía automáticamente
			// Si existe se sobrescribirá
			Writer escritorFicheros = new FileWriter("prueba.txt");
			escritorFicheros.write("Este es el contenido del fichero prueba.txt");
			escritorFicheros.close();

		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

En el código superior podemos observar como **instanciamos nuestra clase de escritura en la primera línea de código**, en donde, además, indicamos la ubicación. En el caso que **no existiera** el fichero, **se crearía** automáticamente.

Si el fichero **existe**, **será sobrescrito**.

En la segunda línea directamente insertamos los **caracteres que deseamos incluir en nuestro fichero.** Los pasamos como parámetro del método "**write()**".

Por último, **cerramos el flujo aplicando el método "close()"**, de esta forma liberaremos los recursos del sistema que han sido ocupados previamente para dicha operación. Una vez **cerrado el flujo** tendremos la **información escrita** en nuestro **fichero (si no cerramos el flujo no se escribe en el fichero)**.

Clases de uso de acceso aleatorio

Instanciación de RandomAcessFile

Utilizaremos la clase **RandomAcessFile** para acceder a ficheros de forma aleatoria. A diferencia de las clases anteriores, permite abrir un fichero en modo lectura y además en **modo lectura-escritura**.

Esta clase permite acceder a un lugar determinado de un fichero y a su vez, leer información o escribirla. Veamos en detalle las distintas **funcionalidades** que nos ofrece dicha **clase**.

Modos de acceso:

• **"r":** modo lectura (read mode) obtendremos un IOException si utilizamos este modo en métodos de escritura.

• **"rw":** modo **lectura y escritura**.

• **"rwd":** modo de lectura y escritura de forma **síncrona**. Se escribirán todas las actualizaciones del contenido del fichero.

• **"rws":** modo de lectura y escritura de forma **síncrona**. Se escribirán todas las actualizaciones del contenido del fichero pero además, se escribirán los **metadatos**.

Veamos ahora cómo instanciar este tipo de objeto y algunos de sus métodos:

RandomAccessFile posee dos **constructores**, en ambos constructores el **segundo parámetro** es el mismo: el **modo de acceso**. Sin embargo, para el **primer parámetro** tenemos la versión del constructor en la que podemos introducir el **objeto File directamente** o, como en el caso del código a continuación, introducir la **ruta del fichero** directamente como String (en este caso, como la ruta está en la raiz del proyecto aparece sólo el nombre del fichero).

```
RandomAccessFile file = new RandomAccessFile("salida.txt","r");
```

Con el método seek() básicamente nos posicionamos en el punto que indiquemos del fichero. Acepta como parámetro un objeto de tipo "long".

```
file.seek(0);
```

Si usamos el método **getFilePointer()**, como bien se define, obtendremos como respuesta un tipo long. Este número es exactamente la **posición del puntero en Bytes.**

```
long filePointer = file.getFilePointer();
```

Método Close automático

*Algo que puede ser realmente interesante es el **cierre automático del flujo** de escritura en la clase RandomAccessFile. Simplemente si instanciamos un nuevo objeto de tipo RandomAccessFile dentro de una sentencia TRY, automáticamente el flujo será cerrado **cuando terminemos las operaciones** que esta **sentencia TRY** englobe con sus llaves.*

Lectura y escritura con RandomAccessFile

Continuamos con nuestra clase RandomAccessFile, a continuación, veremos la forma de **leer y escribir** un **Byte** con esta clase:

Con el método **read()**, podremos leer un byte directamente de nuestro fichero. Devolverá dicho byte a partir de la **posición actual del puntero**.

```
int unByte = file.read();
```

Escritura

Usaremos el método **write()** para escribir un Byte. Dicho Byte será escrito en la posición actual donde se encuentre el **puntero**. Acepta como **parámetro** un **entero** (el **byte a escribir**). En este caso como podemos observar en la imagen, insertaremos la letra "D" en el fichero que corresponde a 68 en la codificación de caracteres ASCII.

```
RandomAccessFile file = new RandomAccessFile("entrada.txt", "rw");
long posicionPuntero = 7;
file.seek(posicionPuntero);
file.write(68);
RandomAccessFile fileLectura = new RandomAccessFile("entrada.txt", "r");
long posicionPunteroLectura = 0;
while (posicionPunteroLectura <= file.length()) {
                        fileLectura.seek(posicionPunteroLectura);
                        System.out.print((char) fileLectura.read());
                        posicionPunteroLectura++;
                    }
```

Una vez **escrito el carácter** en el fichero la posición del **puntero avanzará** una posición.

De la misma forma que hemos realizado una lectura y escritura de un byte, veremos cómo hacerlo de un **array de bytes** en los siguientes fragmentos:

Si en lugar de leer un byte, lo que nos interesa realmente es **leer** una **cantidad determinada de bytes**, podremos realizar una implementación del código que vemos a continuación:

1) En primer lugar, crearemos un **array de bytes, con el tamaño total del fichero** .

```
byte[] arrayBytes = new byte[(int) pruebaFichero.length()];
```

2) A continuación, definimos la **posición del puntero** y el **tamaño de bytes** que queremos **extraer** de nuestro fichero.

```
int inicioPunteroMetodoRead = 0;
int size = 7;
long inicioPunteroLecturaSeek = 11;
                    file.seek(inicioPunteroLecturaSeek);
```

3) Por último, usamos la sentencia **read()**, que en este caso, acepta los siguientes parámetros:

- **el primer parámetro** el **array de bytes** dónde se va a **almacenar** la información,
- **el segundo parámetro**, el **inicio** o posición en la que comenzaremos a leer.
- **El último parámetro** indica la **cantidad de bytes** que deseamos **extraer**.

```
                    int bytesLeidos = file.read(arrayBytes,
inicioPunteroMetodoRead, size);
System.out.println("\n" + size + " bytes leídos, desde el puntero " +
inicioPunteroLecturaSeek + ":");
                    for (int n : arrayBytes) {
                        System.out.print((char) n);
                    }
```

Este método **devuelve** un número entero que representa la **cantidad de bytes** que han sido **leídos**.

Operaciones con buffer

En este caso hablaremos de un conjunto de clases cuyo fin es leer y escribir información **mejorando el rendimiento del sistema** en comparación con nuestras clases aprendidas FileInputStream o FileOutputStream.

Concretamente, nos centraremos en las clases **BufferedInputStream, BufferedOutputStream, BufferedWriter y BufferedReader** que tienen una diferencia notable en relación con las estudiadas previamente.

Primero deberemos entender el concepto de **Buffer**.

Éste, básicamente es un **espacio determinado y temporal** que se aloja en **memoria** para realizar ciertas operaciones.

En las clases que vimos en apartados anteriores, realizábamos operaciones de lectura o escritura capturando Bytes, sin embargo, con las clases con sufijo Buffered el funcionamiento interno será ligeramente distinto ya que se **almacena** en memoria interna **bloques de bytes** completos (**buffer**), y como ya bien sabemos, el **acceso a memoria** es mucho **más rápido que a disco**.

Cada vez que **agotamos esa información** del buffer y se requiere más, se vuelve a **volcar otro bloque de bytes** a la memoria (buffer), de esta forma el rendimiento se ve notablemente mejorado.

A continuación, expondremos un ejemplo de clase **BufferedInputStream:**

```
BufferedInputStream flujoEntradaBuffer = null;
int bufferSize = 4 * 1024;
try {
                flujoEntradaBuffer = new BufferedInputStream(new
FileInputStream(archivoALeer), bufferSize);

                int datos;
while ((datos = flujoEntradaBuffer.read()) != -1) {
                        System.out.print((char) datos);
                }
        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        } finally {
                try {
                        flujoEntradaBuffer.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
```

Definimos el **tamaño del buffer** que se usará posteriormente.

```
int bufferSize = 4 * 1024;
```

A continuación, se instancia nuestro **objeto**, el cual, necesita **dos parámetros**: El primero es un objeto de tipo **FileInputStream** con la **ruta** de nuestro **fichero (o el objeto File directamente)**, y el segundo parámetro es el **tamaño** que deseamos darle a nuestro **buffer**, definido en la primera línea.

```
flujoEntradaBuffer = new BufferedInputStream(
new FileInputStream(archivoALeer), bufferSize)
```

En el bucle "**while**" realizaremos la **lectura** de un bloque de datos siempre y cuando nuestra variable "info" no contenga -1, lo cual, si es así, indicará que el contenido del fichero ha finalizado y que nuestro buffer no tendrá información que proporcionar. Es dentro del bucle donde podremos manipular la información y realizar las operaciones que deseemos.

Por último, simplemente puntualizar la última línea como **cierre y liberación de recursos** de nuestro buffer.

```
flujoEntradaBuffer.close();
```

Documentación API java:

https://docs.oracle.com/javase/8/docs/api/?java/io

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMAS

Flujos

Flujo de datos ("*Streams*"): secuencia ordenada de información que posee un recurso de entrada (flujo de entrada) y un recurso de salida (flujo de salida). **Unidireccionalidad:** se usan sólo para leer o sólo para escribir, pero no ambas situaciones simultaneas.

Según el **tipo de información a tratar,** podemos dividir los Streams en diferentes **categorías**:

- Tratamiento de **ficheros**
- Tratamiento con **Buffer**
- Tratamiento con **Arrays**.
- Tratamiento con **tuberías**.
- Tratamiento con **análisis** (Parsing).
- Tratamiento con **bloques** de información.

Según su **funcionalidad y usabilidad**:

STREAMS		
USABILIDAD	**BYTES (E/S)**	**CARACTERES (E/S)**
Ficheros	**FileInputStream, FileOutputStream**	**FileReader, FileWriter**
Arrays	**ByteArrayInputStream, ByteArrayOutputStream**	**CharArrayReader, CharArrayWriter**
Tuberías	**PipedInputStream, PipedOutputStream**	**PipedReader, PipedWriter**
Buffer	**BufferedInputStream, BufferedOutputStream**	**BufferedReader, BufferedWriter**
Análisis	**PushbackInputStream, StreamTokenizer**	**PushbackReader, LineNumberReader**
Información	**DataInputStream, DataOutputStream, PrintStream**	**PrintWriter**

Tuberías

En Java, proporcionan la capacidad de **ejecutar dos hilos** (threads) ejecutándose en la **misma máquina virtual** (JVM). Esto significa que las tuberías pueden ser **tanto orígenes**, **como destinos** de datos. En **Java**, las partes que se ejecutan deben **pertenecer al mismo proceso** y deben ser **hilos diferentes**. De modo que proceso se relaciona con un mismo lugar de memoria. En otras palabras, para comunicar 2 tuberías, una de entrada y otra de salida (por ejemplo), deberán de ejecutarse bajo el mismo proceso y con 2 threads independientes.

Ejemplo:

escribir datos desde un hilo y leerlos desde otro, dentro del mismo proceso:

```
package streamscontuberias;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import java.util.logging.Level;
import java.util.logging.Logger;
public class StreamsConTuberias {
   public static void main(String[] args) throws IOException {
      /*
      Instanciamos los objetos para entrada y salida de flujos
      El primero de ellos hace referencia a un objeto de salida,
      el cual, podría ser por ejemplo una escritura de un fichero,
      pero en este caso simplemente simularemos la escritura de la frase:
      "Hola chavales!".
       */
      final PipedOutputStream salida = new PipedOutputStream();
      /*
      instanciamos la tubería de entrada pasándole por parámetro la salida
      creada previamente. De esta forma la "tubería" quedará conectada
      y tendremos acceso al fichero de salida que está realizando la escritura.
       */
      final PipedInputStream entrada = new PipedInputStream(salida);
      String textoSalida = "Hola chavales!";
```

```
        /*
         Creamos 2 hilos:
            hilo1 para escribir
            hilo2 para leer
        En el hilo1 escribimos los bytes referenciados
         */
         Thread hilo1 = new Thread(new Runnable() {
            @Override
            public void run() {
               try {
                  /*
                    Dentro del hilo escribimos los bytes en la salida
                  */
                   salida.write(textoSalida.getBytes());
               } catch (IOException ex) {
                    Logger.getLogger(StreamsConTuberias.class.getName()).log(Level.SEVERE, null,
ex);
               }
            }
         });
        /*
        En el hilo2 leremos "al mismo tiempo" en la misma tubería
        el objeto PipedInputStream nombrado entrada
         */
         Thread hilo2 = new Thread(new Runnable() {
            @Override
            public void run() {
               try {
                  /*
                 Como en entrada tenemos como argumento salida,
                 podremos leer lo escrito en salida:
                 leemos directamente del PipedInputStream cuya variable
```

```
            definimos “entrada”, referenciada directamente a la salida,
            por lo cual tenemos acceso a ella.
            Básicamente lo que hacemos es leer directamente Byte a Byte
            del fichero de salida al mismo tiempo que se está escribiendo.
             */
             int unByte = entrada.read();
              while (unByte != -1) {
                 System.out.print((char) unByte);
                 unByte = entrada.read();
              }
            } catch (IOException e) {
            }
        }
     });
     /*
     Sólo nos queda lanzar los dos hilos
     ejecutamos ambos hilos, teniendo en cuenta que “hilo2” va a tener acceso
     de lectura a “hilo1” y, por lo tanto, va a mostrar por pantalla justo
     lo que “hilo1” está escribiendo.
      */
     hilo1.start();
     hilo2.start();
   }
}
```

Flujos basados en arrays

- Acceso basado en caracteres: **CharrArrayReader, CharArrayWriter**.
- Acceso basado en bytes: **ByteArrayInputStream**, **ByteArrayOutputStream**.

Caracteres

CharrArrayReader y CharrArrayWriter

CharrArrayReader: leer el contenido de un array de caracteres (char) como un Stream de caracteres. Esta clase nos será útil en casos que tengamos **información en un array de caracteres** y necesitemos **pasarlo a** algún componente, el cual solo pueda **ser leído desde una clase Reader** o una subclase.

Instanciaremos un **CharArrayWriter** y escribiremos un texto. Después **pasaríamos por parámetro** dicho objeto para construir un nuevo elemento, concretamente, sería construir un objeto de la clase **CharArrayReader**. De esta forma, tendríamos en esta clase el contenido de todo el objeto CharArrayWriter. Después podríamos ir **leyendo el objeto "charArrayReader" byte a byte** para realizar lo que se desee en la parte del comentario "BLOQUE". En este caso se ha **imprimido por pantalla** el array leído desde el objeto CharArrayReader.

Por último ejecutaríamos el método close() liberando los recursos, tal y como estamos habituados a hacer.

```
package escrituralecturaarraycaractereschararrayreader.writer;

import java.io.CharArrayReader;

import java.io.CharArrayWriter;

import java.io.IOException;

import java.util.logging.Level;

import java.util.logging.Logger;

public class EscrituraLecturaArrayCaracteresCharArrayReaderWriter {

    public static void main(String[] args) {

        CharArrayWriter writer = new CharArrayWriter();

        CharArrayReader reader;

        /*

        en data asignamos la información que leeremos

         */

        int data = 0;

        try {
```

```
            /*
            Escribimos en el escritor de array de caracteres
             */
            writer.write("Esto es escrito en un objeto CharArrayWriter");
            /*
            Usamos CharArrayReader a partir del mismo CharArrayWriter writer
           pasándolo como parámetro, convertido a array de caracteres
             */
            reader = new CharArrayReader(writer.toCharArray());
            /*
            leemos el primer byte, para luego seguir leyendo mientras
           no lleguemos al final del array de caracteres
             */
            data = reader.read();
            while (data != -1) {
                // BLOQUE
                System.out.println((char) data);
                // incrementamos el caracter leido para la próxima iteración
                data = reader.read();
            }
            /*
            Como no usaremos más el flujo de lectura de caracteres lo cerramos
             */
            reader.close();
        } catch (IOException ex) {
            Logger.getLogger(EscrituraLecturaArrayCaracteresCharArrayReaderWriter.class.getName()).l
og(Level.SEVERE, null, ex);
        }
        // Si no lo usamos más lo cerramos
        writer.close();
    }
}
```

Clases de análisis para flujos de datos

PushBackInputStream, StreamTokenizer, PushbackReader, LineNumberReader.

PushBackInputStream y PushbackReader: análisis de datos previo de un InputStream.

En algunas ocasiones se necesita **leer algunos bytes con anticipación** para saber qué se aproxima, para así poder interpretar el byte actual.

Estos bytes leídos con anterioridad (ya leídos con el método read()), serán **de nuevo 'empujados' a la secuencia**, para que posteriormente, los leamos cuando se haga de nuevo read(). Es como invalidar el último read() para volver a leer ese byte dependiendo del valor obtenido, es decir, nos da una segunda oportunidad de leer un byte, pero en este caso ya lo habremos podido leer.

- **PushbackInputStream: byte** a byte.
- **PushbackReader: caracteres**.

En el siguiente código: un byte podrá ser leídos con anterioridad, y más tarde devuelto a la secuencia o al flujo de bytes:

```
package retrocesodereaderpushbackreader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PushbackReader;
import java.io.Writer;
import java.util.logging.Level;
import java.util.logging.Logger;
public class RetrocesoDeReaderPushbackReader {
   public static void main(String[] args) {
      // Creamos un fichero para luego leerlo con PushbackReader
      File fichero = new File("fichero");
      try {
         fichero.createNewFile();
         Writer escritor = new FileWriter(fichero);
```

```
            escritor.write("Este es el contenido del fichero");
        } catch (IOException ex) {
            Logger.getLogger(RetrocesoDeReaderPushbackReader.class.
                    getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                escritor.close();
            } catch (IOException ex) {
                Logger.getLogger(RetrocesoDeReaderPushbackReader.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }
        try {
            // instanciamos un objeto PushBackReader
            PushbackReader pushBackReader = new PushbackReader(
            /*
             para ello necesitamos a su vez instanciar un objeto FileReader
            indicando la ruta del fichero a tratar
            la primera letra del fichero es “E”.
            */
                    new FileReader(fichero));
            /*
            usamos el método read() y justo
           después mostramos por pantalla el byte leído haciendo casting de
           “char”.
           mostramos por pantalla la letra “E”.
            */
            int data = pushBackReader.read();
            System.out.println((char) data);
            /*
            Justo en la siguiente línea nos aparece un método nuevo unread(data).
           Es el método clave de esta clase definida. Este método devuelve al
```

```
            stream de datos el byte que hemos leído con anterioridad, de tal
            forma que si volvemos a hacer un read() y mostramos por pantalla,
            obtendremos el mismo byte que antes. En este caso la letra “D”.
             */
             pushBackReader.unread(data);
             data=pushBackReader.read();
             System.out.println((char)data);



        } catch (FileNotFoundException ex) {
            Logger.getLogger(RetrocesoDeReaderPushbackReader.class.
                    getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
                Logger.getLogger(RetrocesoDeReaderPushbackReader.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
}
```

Clases de análisis para flujos de datos

StreamTokenizer

Un token viene a estar relacionado con un “fragmento” o “**trozo**” … La clase StreamTokenizer tiene la capacidad de **analizar el fichero por** ‘fragmentos’. Dichos fragmentos más tarde tendremos que evaluarlos y comprobar si son palabras o números. Es capaz de reconocer identificadores: números, comillas, espacios, etc., lo cual nos puede ser de gran utilidad en aplicativos de análisis de ficheros según su tipología sea números o caracteres.

Ejemplo

```
package analisisfragmentosficherostreamtokenizer;
/*
 La clase StreamTokenizer tiene la capacidad de analizar el fichero por «trozos»
o «fragmentos». Evaluaremos dichos fragmentos y comprobaremos si son palabras
o números. Reconoce identificadores: números, comillas, espacios, etc.,
lo cual nos puede ser de gran utilidad al analizar ficheros dependiendo de si
su tipología sea de números o de caracteres.
 */
import java.io.IOException;
import java.io.StreamTokenizer;
import java.io.StringReader;
import java.util.logging.Level;
import java.util.logging.Logger;
public class AnalisisFragmentosFicheroStreamTokenizer {
    public static void main(String[] args) {
        /*
         instaciamos el objeto StreamTokenizer usando el constructor por medio
        del cual le pasamos un StringReader.
         */
        StreamTokenizer streamTokenizer = new StreamTokenizer(new StringReader(""
                + "Este es el texto de un StreamTokenizer"));
        /*
        algunos de sus métodos estáticos nos dan información de la tipología de
```

```
        los distintos Tokens:
        - TT_EOF: indica el final del fichero (End Of File)
        - TT_EOL: indica el final de la línea (End Of Line)
        - TT_WORD: indica que el token es de tipo palabra, conjunto de letras.
        - TT_NUMBER: indica que el token evaluado es un número o una asociación
        de ellos.
         */
        try {
            while (streamTokenizer.nextToken() != StreamTokenizer.TT_EOF) {
                // si el token es de tipo palabra, mostraremos por pantalla dicha palabra.
                if (streamTokenizer.ttype == StreamTokenizer.TT_WORD) {
                    System.out.println(streamTokenizer.sval);
                    // si es de tipo número, mostraremos por pantalla el número.
                } else if (streamTokenizer.ttype == StreamTokenizer.TT_NUMBER) {
                    System.out.println(streamTokenizer.nval);
                    // si es de tipo final de línea se imprimirá retorno de carro.
                } else if (streamTokenizer.ttype == StreamTokenizer.TT_EOL) {
                    System.out.println();
                }
            }
        } catch (IOException ex) {
            Logger.getLogger(AnalisisFragmentosFicheroStreamTokenizer.class.getName()).log(Level.SE
VERE, null, ex);
        }
    }
}
```

Clases de análisis para flujos de datos

LineNumberReader

Existen varios **métodos** clave para esta clase:

- **getLineNumber**(): este método como su nombre indica, nos devolverá el número de la línea **en la que estemos** actualmente leyendo.
- **setLineNumber**(): este método es realmente interesante ya que se le puede pasar como parámetro un entero, y **se convertirá en la línea actual**.
- **readLine**(): nos devolverá un conjunto de caracteres al cual se le debe asignar a una variable String.
- **read**(): lee un buffer.

Veamos un **código con el uso de esta clase**:

```
package contadorlineaslinenumberreader;
/*
LineNumberReader es hija de BufferedReader.
Almacena y cuenta el número de líneas leídas de caracteres.
Está orientada a trabajar y analizar líneas completas.
Empieza leyendo por la primera línea con el contador a 0 y cada vez que
encuentra un retorno de carro incrementa su valor en +1.
 */
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.logging.Level;
import java.util.logging.Logger;
public class ContadorLineasLineNumberReader {
    public static void main(String[] args) {
        try {
            /*
             Se instancia LineNumberReader con un nuevo objeto FileReader, el cual
            es instanciado con un fichero que se ha agregado al proyecto.
```

```
         */
        LineNumberReader lineNumberReader
              = new LineNumberReader(new FileReader("prueba.txt"
              ));
        // quedará almacenado el contenido de la primera línea con readLine()
        String line = lineNumberReader.readLine();
        /*
         mientras se tengan líneas que recorrer, se irá mostrando por
        pantalla tanto el número de la línea en la que se está posicionado,
        como su contenido, a través de: getLineNumber() y
        System.out.println(line). De esta forma iremos mostrando el
        contenido del fichero completo línea tras línea.
         */
        while (line != null) {
           System.out.println("Línea número "
                 + lineNumberReader.getLineNumber()
                 + ": "
                 + line);
           line = lineNumberReader.readLine();
        }
        lineNumberReader.close();
      } catch (FileNotFoundException ex) {
            Logger.getLogger(ContadorLineasLineNumberReader.class.getName()).log(Level.SEVERE,
null, ex);
      } catch (IOException ex) {
            Logger.getLogger(ContadorLineasLineNumberReader.class.getName()).log(Level.SEVERE,
null, ex);
      }
   }
}
```

Clases para el tratamiento de información

Nos permiten **procesar tipos primitivos de Java**: int, float, long etc.

DataInputStream

Nos ofrece una gran ventaja en comparación con InputStream:

Veamos un **ejemplo** de uso de **DataInputStream y DataOutputStream** con tipos de **datos primitivos**:

```
package lecturatiposprimitivosdatainputstream;
/*
Útil para  lectura de tipos primitivos.
Se suele usar la clase DataInputStream para leer ficheros que previamente han
sido escritos con DataOutputStream.
Hereda métodos de InputStream.
Si queremos leer byte a byte con el método read(), evidentemente lo tendremos
también disponible, así como la lectura con un array de bytes.
Atencion: cuando leemos tipos de datos primitivos no hay forma de distinguir la
lectura de un número “-1” a la lectura de fin de flujo que es también -1,
por lo tanto, es muy importante en este tipo de lectura saber qué tipo de datos
vamos a leer y qué orden llevan.
 */
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
public class LecturaTiposPrimitivosDataInputStream {
    public static void main(String[] args) {
```

```
try {
    // vamos a introducir datos en el fichero
    /*
    para leer de un fichero por medio de la clase DataInputStream,
   antes hemos tenido que realizar la escritura ordenada del fichero y
   debemos conocer el tipo de datos y la cantidad de ellos que ha sido
   insertada. Usamos DataOutputStream.
   El objeto es instanciado en su constructor con un FileOutputStream
   con la ruta del fichero que vamos a escribir.
     */
    DataOutputStream salida = new DataOutputStream(
            new FileOutputStream("datos.bin"));
    /*
    Escribimos en el fichero distintos tipos primitivos de Java usando
   diferentes métodos:
   • writeInt() : solo aceptara entero como parámetro.
   • writeFloat() : solo aceptara float como parámetro.
   • writeDouble() : solo aceptara double como parámetro.
     */
    salida.writeInt(10);
    salida.writeFloat(10.10F);
    salida.writeDouble(10.1234);
    // cerramos nuestro flujo y liberamos recursos.
    salida.close();
    // ahora vamos a crear el flujo de lectura de datos primitivos
    DataInputStream dataInputStream = new DataInputStream(
            new FileInputStream("datos.bin"));
    int entero = dataInputStream.readInt();
    float numFloat = dataInputStream.readFloat();
    double numDouble = dataInputStream.readDouble();
    dataInputStream.close();
    System.out.println("El número entero es: " + entero);
```

```
            System.out.println("El número float es: " + numFloat);
            System.out.println("El número double es: " + numDouble);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(LecturaTiposPrimitivosDataInputStream.class.getName()).log(Level.SEVERE
, null, ex);
        } catch (IOException ex) {
            Logger.getLogger(LecturaTiposPrimitivosDataInputStream.class.getName()).log(Level.SEVERE
, null, ex);
        }
    }
}
```

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMAS

Ficheros XML

Acceso a datos con DOM Y SAX

Tanto **DOM** como **SAX** son estándares, herramientas que nos ofrecen la posibilidad de **lectura de ficheros XML**. V**erifican si sintácticamente son ficheros válidos**. Son los llamados "**parsers**" o **analizadores**. Algunas de características y diferencias entre DOM y SAX:

• La **ventaja** que tenemos con el sistema **DOM** es que una vez **introducimos el fichero** HTML o XML, obtenemos el **árbol** ya formado de los nodos y demás objetos, **preparado para trabajar**. Pero es más **lento** y **menos versátil**. **SAX es más rápida**, pero **menos potente** que DOM. Con **SAX** necesitamos introducir **líneas de programación** para obtener partes determinadas de los ficheros. Nos ofrece **mayor** nivel de **funcionalidad y versatilidad**.

• El funcionamiento de DOM y SAX es muy diferente:

- DOM carga el fichero completo, tenemos todo el árbol del fichero disponible, pero ocupa mucha **más memoria** que SAX.
- **SAX tiene en memoria sólo** la parte del **nodo** o el **evento actual**.

En resumen:

- Usar **SAX** para **recorrer secuencialmente** los elementos del fichero XML y realizar ciertas operaciones.
- Usar **DOM** cuando tengamos el **objetivo claro** sobre el que queremos trabajar, a partir de un **árbol creado** en memoria.

Características de ambos sistemas:

SAX	DOM
Basado en eventos	Búsqueda de tags hacia delante y hacia detrás.
Va analizando nodo por nodo Secuencial	Estructuras de árbol Nodos en modo árbol Análisis del fichero completo
Sin muchas restricciones de memoria Menos memoria en uso no carga la totalidad del fichero solo si necesitamos fragmentos de documento	Carga el fichero en memoria Ocupa más memoria
Rapidez en tiempo de ejecución.	Más lento en tiempo de ejecución.
Menos potente	Más potente
Es solo de lectura	Se pueden insertar o eliminar nodos Buena opción para editar Múltiples procesos

Conversión de ficheros XML

Parsers (analizadores) de ficheros XML hay muchos. Pero ahora nos centraremos en la paquetería: "**javax.xml.parsers**".

Ejemplo de un "**parser" de tipo DOM.** La estructura se cargará en memoria y tendremos disponible el fichero completo.

```
package parserxml_dom;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
public class ParserXML_DOM {
  static DocumentBuilder builder;
  public static void main(String[] args) {
    // Instanciamos la clase DocumentBuilderFactory
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    /*
    Establecemos el atributo de validación como "true" para
    asegurarnos que el fichero que se cargue esté bien validado.
    */
    factory.setValidating(true);
    /*
    Se hace un set también, al atributo de "ignorar los elementos que
    contengan espacios en blanco", a "true".
    */
    factory.setIgnoringElementContentWhitespace(true);
    try {
      /*
```

```
      Creamos un objeto DocumentBuilder por medio de la factoría creada
      previamente.
      */
      builder = factory.newDocumentBuilder();
    } catch (ParserConfigurationException ex) {
      Logger.getLogger(ParserXML_DOM.class.getName()).log(Level.SEVERE, null, ex);
    }
    // Instanciamos un nuevo fichero indicando la ruta del fichero a analizar.
    File file = new File("ejemplo.xml");
    try {
      /*
      Cargamos el fichero completo con el método builder.parse(file), y se
      asigna a un objeto de tipo Document. De esta forma quedará almacenado, y
      podremos realizar diferentes acciones con dicho objeto en las líneas
      siguientes.
      */
      Document doc = builder.parse(file);
    } catch (SAXException ex) {
      Logger.getLogger(ParserXML_DOM.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
      Logger.getLogger(ParserXML_DOM.class.getName()).log(Level.SEVERE, null, ex);
    }
  }
}
```

Ejemplo de cómo instanciar y trabajar con un parser (analizador) de XML, de tipo SAX.

```
package parserxml_sax;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class ParserXML_SAX {
  static File file;
  public static void main(String[] args) throws SAXException {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setValidating(true);
    SAXParser saxParser;
    try {
      saxParser = factory.newSAXParser();
      file= new File("ejemplo.xml");
      /*
      Se instanciará, en el método parse, un Handler que será el
      responsable de ejecutar ciertas operaciones como iniciar elementos,
      operaciones con nodos, inicio/fin de documento, etc.
      Es en la definición del Handler en donde debemos indicar las
      operaciones que realice nuestro analizador de código.
      */
      saxParser.parse(file, new DefaultHandler());
    } catch (ParserConfigurationException ex) {
      Logger.getLogger(ParserXML_SAX.class.getName()).log(Level.SEVERE, null, ex);
```

```
    } catch (IOException ex) {
      Logger.getLogger(ParserXML_SAX.class.getName()).log(Level.SEVERE, null, ex);
    }
  }
}
```

Procesamiento de XML
XPath

XPATH se usa para realizar **búsqueda de información** a través de un documento **XML**. Es una **recomendación** oficial del consorcio del **World Wide Web** (**W3C**).

Se utiliza para **recorrer elementos y atributos** de un documento XML, y proporciona varios tipos de **expresiones** que pueden usarse para consultar información relevante.

Características principales de XPATH

- **Definición de estructuras**: define las distintas partes de un documento XML como un **elemento**, **atributos**, **textos**, **instrucciones** de procesamiento, **comentarios** y **nodos** del documento.
- **Expresiones**: XPATH posee expresiones potentes para el **manejo de ficheros**, como por ejemplo **seleccionar nodos** o **listas de nodos** en ficheros XML.
- **Funciones** estándar: posee **librería** muy completa de funcionalidades estándar de manipulación de Strings, valores numéricos, fechas, comparaciones, secuencias, valores booleanos...

Cómo hacer uso de esta librería XPath:

Al usar la librería XPATH:

- Importaremos los **paquetes** relacionados con dicha librería.
- Crearemos un **objeto** de la clase **DocumentBuilder.**
- Cargaremos un **fichero** o un flujo de datos.
- Crearemos un **objeto XPATH** y una **expresión**.
- Realizaremos una **compilación de dicha expresión** con el método **Xpath.compile()** y obtendremos una **lista de los nodos evaluando la expresión** previamente compilada usando **Xpath.evaluate().**
- Realizaremos una **iteración** por lo general de la **lista de nodos**.
- **Examinaremos los atributos**.
- **Examinaremos los sub elementos**.

Ejemplo de uso de librería Xpath:

```
package com.mycompany.usolibreriaxpath2;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
public class UsoLibreriaXPath2 {
  public static void main(String[] args) {
    File file = new File("ejemplo.xml");
    DocumentBuilderFactory dBFactory = DocumentBuilderFactory.newInstance();
    DocumentBuilder dBuilder;
    try {
      dBuilder = dBFactory.newDocumentBuilder();
      /*
      Creamos un Document con el método parse (analiza) del DocumentBuilder
      con el que podremos analizar el documento .xml
       */
      Document doc = dBuilder.parse(file);
      doc.getDocumentElement().normalize();
```

```
        // La instanciación del objeto Xpath la realizaremos de la siguiente forma:
      XPath xPath = XPathFactory.newInstance().newXPath();
      String expresionXPath = "/pizzas/pizza";
      // creamos una lista de nodos para la clase /pizza
      NodeList nodeList = (NodeList) xPath.compile(
          expresionXPath).evaluate(
              doc, XPathConstants.NODESET);
      // recorremos los nodos para trabajar con el xml:
      for (int i = 0; i < nodeList.getLength(); i++) {
        Node nNode = nodeList.item(i);
        System.out.println("\nCurrent Element: " + nNode.getNodeName());
        if (nNode.getNodeType() == Node.ELEMENT_NODE) {
          Element element = (Element) nNode;
          System.out.println("Nombre de la pizza: "
              + element.getAttribute("nombre")
              + "\nPrecio: " + element.getAttribute("precio")
            /*
             podemos seguir iterando en nodos agregando nueva
            expresion XPath...
             */
            );
        }
      }
      /* multicatch para en este caso ahorra líneas de código.
      Se aconseja tratar cada try-catch por separado...
       */
            } catch (ParserConfigurationException | SAXException | IOException |
XPathExpressionException ex) {
      Logger.getLogger(UsoLibreriaXPath2.class.getName()).log(Level.SEVERE, null, ex);
    }
  }
}
```

Excepciones

Breve introducción a las mismas a modo genérico:

Excepciones en Java y tipos.

Una excepción no es más que un evento que ocurre durante la ejecución de un programa, y que interrumpe el flujo del mismo por algún motivo.

Las **excepciones se dividen en 3 categorías**:

• Excepciones **con chequeo** (**chequed exceptions**): Notificadas por el compilador en tiempo de **compilación**, no pueden ser ignoradas, y **fuerzan al programador a manejarlas**.

• Excepciones **sin chequeo** (**unchequed exceptions**): En **tiempo de ejecución**, llamadas también **RuntimeExceptions**. Se incluyen aquí también, **errores de programación** o cuando se ha **usado mal una API** de código, por ejemplo. Comentar también que este tipo de excepciones son **ignoradas en tiempo de compilación**.

• **Errores**: no son del todo excepciones, escapan del control del usuario o del propio desarrollador. Los errores generalmente se ignoran en el código porque rara vez se puede hacer algo al respecto. Un **ejemplo** de error es **Stack overflow** si hay un desbordamiento de pila, y difícilmente vamos a poder hacer algo para solucionar este problema. Este tipo de errores son **ignorados en tiempo de compilación** también.

Métodos más importantes en el manejo de excepciones con ficheros XML

• **getMessage():** devuelve un mensaje detallado sobre la excepción que se acaba de lanzar. El mensaje es **instanciado en el constructor** de la clase **Throwable**.

• **getCause():** devuelve la causa representada en un **objeto Throwable**.

• **toString():** devuelve el nombre de la clase y se le **concatena el resultado de getMessage()**.

• **printStackTrace():** imprime el resultado del **método toString()** junto con el **error de sistema** que devuelve la **pila.** Es **más completo** que los anteriores ya que envuelve a todos ellos.

• **getStackTrace():** devuelve un **array** con cada uno de los elementos de la pila. El **elemento 0** del array representa el elemento **más alto** de la pila.

• **fillInstackTrace(): rellena** la **pila del objeto Throwable** con la **pila actual**. Le añade cualquier **información previa** en el seguimiento de la misma.

Para poder **detectar y tratar una excepción** necesitamos incluir un bloque de código **try-catch**. Este bloque se coloca alrededor del código que pudiera generar una excepción. El código incluido dentro de un bloque try/catch se conoce como **código protegido**.

Sintaxis de la sentencia try-catch:

```
try {
  // Código protegido
} catch (ExceptionName e) {
        // Operaciones tras capturar excepción
        }
```

El código que es propenso a excepciones se coloca a continuación de la sentencia try. Cuando se lanza una excepción, es capturada por el bloque catch asociado a ella. **Cada bloque try** puede ir seguido de un **bloque catch** o de un bloque **finally**.

Un bloque catch como podemos observar implicará **declarar un tipo de la excepción** que queramos **capturar** Un bloque **finally** lo encontraremos justo **después del bloque try** o después del **catch**. Es una parte de código que se **ejecutará siempre**, independientemente si pasa por éstos. Este tipo de bloques suelen usarse para **labores de limpieza** o **liberación de recursos** de memoria, por ejemplo.

Para aquellas sentencias try-catch que sean **obligatorias**, nuestro entorno de desarrollo nos lo notificará.

El IDE nos subrayará una operación, y si situamos el cursor encima de ésta, nos avisará que hay una excepción que no está siendo controlada, además nos informa de que tipo es. Si nos fijamos en la parte izquierda de esa misma línea, en el margen de nuestro editor de texto observaremos una indicación. Si hacemos clic en ella tendremos varias opciones.

Concretamente, una vez hacemos clic en la pequeña bombilla roja, tendremos:

• Añadir la **excepción a la definición del método**: con esta opción lo que estaremos haciendo será lanzar la excepción a un **nivel superior**. De esta forma se irá **lanzando la excepción de un nivel a otro** hasta que alguno de ellos decida tratarla.

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

• Rodear con **sentencia try/catch**: tal y como hemos visto previamente, rodearíamos el código con la estructura básica try/catch. De esta forma el **código quedará protegido** y si se lanza dicha **excepción, será capturada y tratada**.

```
try {
dBuilder = dbFactory.newDocumentBuilder();
} catch (ParserConfigurationException e) {
        e.printStackTrace(); }
```

Ya estudiado el procedimiento de cómo capturar una excepción desde un bloque de código try/catch, ahora veremos algunos otros ejemplos de diferentes **excepciones**, que lanzan las operaciones que ejecutamos con nuestros **parsers**:

Dispondremos con el IDE de una línea de código en la que:

un objeto de la clase DocumentBuilder está ejecutando el método parse() y le está pasando como parámetro un fichero. Bien, el IDE nos **indica que hay un error**, algo falta, una excepción que no está siendo manejada.

Todos los IDEs tienen una combinación de **teclas para mostrar la documentación** de Java sobre el método en el que se está, en este caso, parse(). En esta documentación, en el **apartado de Throws**, podremos ver qué **excepciones podría lanzar** el método que se está ejecutando. De esta forma, es una muy buena práctica pensar y cubrir las distintas opciones con bloques catch, **capturando cada una de estos posibles lanzamientos** de excepciones.

Por último, comentar, que cuando estemos realizando bastantes operaciones del estilo, en las cuales, se vean envueltas un **número considerable de excepciones**, es una buena práctica poner **un bloque try**, añadir las líneas necesarias, y a continuación los **bloques catch anidados unos con otros**, para así evitar tener que ir escribiendo contínuamente bloques try/catch.

Ejemplo:

```
try {
// operaciones parse...
} catch (ParserConfigurationException e) {
        e.printStackTrace();
} catch (IOException e) {
        e.printStackTrace();
} catch (XPathExpressionException e) {
        e.printStackTrace();
} catch (SAXException e) {
        e.printStackTrace();
}
```

Englobamos todos los bloques catch de una **serie de operaciones de análisis o "parsing"**, teniéndolas **mejor distribuidas y manejadas** de esta forma.

Pruebas y documentación

JUnit

Un **test** es una **pieza de Software que ejecuta otra pieza de Software**. **Valida** si los resultados de un código están en el **estado** que se espera, o **ejecuta la secuencia** esperada de operaciones o eventos. Ayudan al programador a verificar que un fragmento de **código es correcto**.

Junit es un framework que usa **anotaciones** para **identificar diferentes tests**. Una **prueba unitaria** Junit es realmente un **método** que está en el **interior de una clase llamada Test class**. Para definir que un **método** forma **parte de un test** se tendrá que añadir la **anotación @Test sobre la cabecera del método**.

Si queremos disponer de las **librerías necesarias** para poder realizar nuestras pruebas unitarias, podríamos escribir la anotación "@Test" ya que es una palabra clave, y nuestro IDE entenderá que queremos introducir librerías de Junit por ser el framework más extendido de Unit Testing.

Nos dará a elegir entre las versiones estables del momento y elegiremos la que más nos convenga.

Otra opción en **otro tipo de proyectos** (proyectos web por ejemplo) sería **añadir una nueva dependencia** maven con la librería correspondiente **Junit y su versión**.

Existen algunas anotaciones y algunos métodos muy interesantes que debemos tener en cuenta para la escritura de test unitarios, como por ejemplo:

• Anotación **@Before**: al inicio de la clase se definirá un método. Su utilidad será **instanciar la mayor parte de las variables que vamos a necesitar** para los test. Siempre que ejecutemos un test unitario, **antes se ejecutará el código** de este método con anotación **@Before**. Hará un set up de los datos de la clase a testear.

• Anotación **@After**: Con esta anotación, se definirá un método cuyo código **se ejecutará**, siempre después de **finalizar cualquier test** dentro de nuestra clase. El método podría llamarse **tearDownClass**, o demoler la clase, ya que se pretende eliminar lo que ya no es necesario, limpiar...

Ejemplo:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
public class NewEmptyJUnitTest {
  static private int variable;
  public NewEmptyJUnitTest() {
  }
  // el código pasa antes por @BeforeAll:
```

```
  @BeforeAll
  public static void instanciarVariablesTests() {
    // se realiza un set up de datos, este es un ejemplo:
    variable = 2;
  }

  // realizamos nuestro test:
  @Test
  public void test1() {
    // Aquí implementamos nuestro código para testear

  }

  /*
   Se utiliza al final de cada test, sobre todo para liberar recursos,
  limpiar basura, ahorrar recursos...
   */
  @AfterAll
  public static void demolerClase() {
  }

  // otra manera de usar @Before
  @BeforeEach
  public void setUp() {
  }

  // otra manera de usar @After
  @AfterEach
  public void tearDown() {
  }
}
```

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Conectores a base de datos

Conector

Serie de **clases y librerías** que **unen** la capa de nuestra **aplicación** con la capa de **base de datos**. Este **punto intermedio** es nuestro conector, y es necesario para conectarnos a la base de datos y realizar consultas.

Desfase objeto-relacional

Se refiere a cuando **surgen discrepancias: bases de datos tienen naturalezas** distintas en comparación a la aplicación que se trabaja con programación **orientada a objetos**. Esto se llama: desfase objeto-relacional.

Aspectos importantes del desfase:

- **Diferencia entre los datos**: diferencias en los datos:

  En la BDD **relacional** son **datos simples;**

  En la **orientada a objetos** son **objetos complejos**.

- **Realizar una traducción**: Realizar distintos **diagramas**: traducción desde los objetos de la aplicación Java a la base de datos relacional. **Entidades distintas** representen la **misma unidad**.

Protocolos de acceso a base de datos

Realmente, un conector o **driver** es una serie de clases implementadas (API) que facilitan la **conexión a la base de datos** asociada.

Basándonos en el lenguaje **SQL**, disponemos de **dos protocolos** de conexión:

- **JDBC** (Java Database Connectivity) (**Sun)**.
- **ODBC** (**Open** DataBase Connectivity) (**Microsoft**) B*asado en la conexión con bases de datos puras SQL. API desarrollada en* ***lenguaje C.***

Otros protocolos:

De Microsoft también tenemos:

- ***ADO.NET***
- ***ADO.NET + LINQ***
- ***OLE/ADO DB***

Una aplicación debe tener asociado siempre un conector. Cuando estamos desarrollando una aplicación e introducimos un conector no tenemos que conocer los aspectos técnicos, ni cómo funcionan en su interior dichas bases de datos, sino sólo en **cómo realizar la comunicación** y de **cómo funcione** nuestra aplicación.

El **conector interpretaría** de una forma u otra **dependiendo de la base de datos** asociada.

Si nuestra aplicación necesita información de una base de datos, **utilizando** la **librería correspondiente** e indicando las **configuraciones de acceso** a cada base de datos, tendremos el acceso sin preocuparnos del lenguaje interno de cada una.

Conexiones JDBC:
Componentes y tipos

Componentes JDBC

Son **cuatro:**

- **API JDBC**:
    - librerías y clases que nos facilitan:

        1- **acceso** a las bases de datos relacionales.

        2- **consultas** a la base de datos.

    java.sql y **javax.sql**.
- **Paquete de pruebas JDBC**:

    Valida si un **driver** pasa los **requisitos** previstos por JDBC.
- **Gestor JDBC**:

    Realiza la **unión** (conexión) **aplicación** - **driver** apropiado JDBC.

    Hay **dos formas** de conexión:

    • **Directa**.

    • Con **pool** de conexiones.
- **Puente JDBC-ODBC**: facilita el **uso de los drivers ODBC** como si estuviéramos trabajando con JDBC.

2 tipos de Arquitectura de conexión con JDBC

• **En dos capas**: nuestra aplicación se conectará a la BDD a través de **un driver**. Driver y aplicación estarán **en el mismo sistema o máquina**.

Será ideal para una **aplicación simple** que **no requiere muchos recursos** ni se prevé que vaya a tener multitud de consultas. Se puede instalar el **conector** (driver) en la **misma máquina del cliente** realizando las labores de **traducción** y comunicándose **directamente con la base de datos**.

Las capas serán:

1- la **aplicación junto con el driver** en el sistema o máquina

2- y la **base** de datos.

• **En tres capas**: La aplicación envia instrucciones a una **capa intermedia (driver), a modo de traductor** (***middleware*** o software intermedio). La capa intermedia o *middleware* **cogerá** la información y la **enviará** a la base de datos correspondiente **traduciendo los comandos** que la aplicación haya enviado. Es más aconsejada para **aplicaciones web (**e-comerce), en la que se aísla el driver del sistema que contiene la aplicación, **no** teniendo que hacer esta **traducción** de comandos **en el sistema** o máquina donde se hace la petición. De esta forma es **más rápido** el acceso a la base de datos y la respuesta a la aplicación en el lado del cliente (aplicación o sistema).

Útil para aplicación (escritorio o web) cuyo propósito sea gestionar una **cantidad grande de consultas** y sea necesario balancearlas, incluso con algún tipo de caché.

Las capas serán:

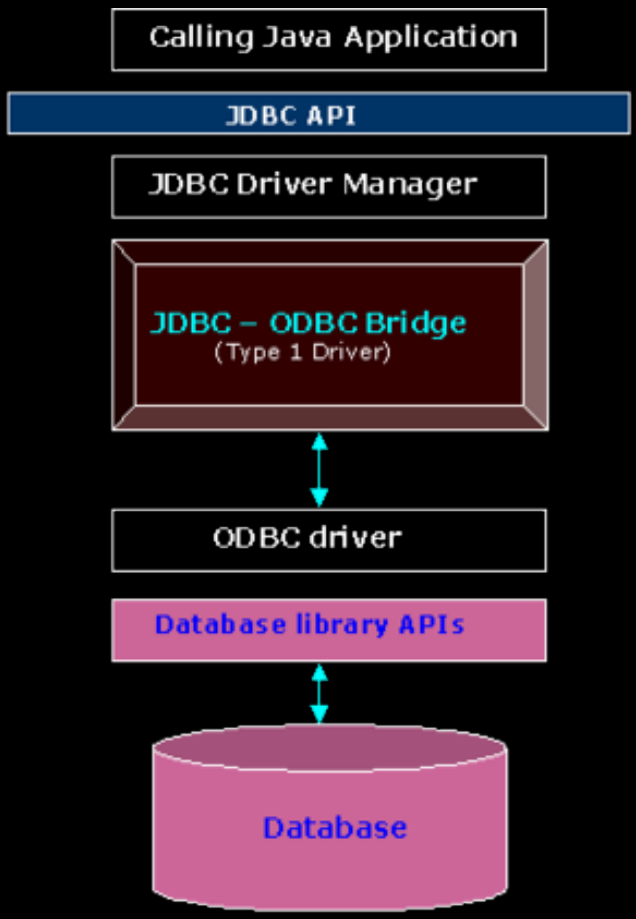
1- la **aplicación** en el sistema o máquina,

2- el **driver** *middleware o traductor,*

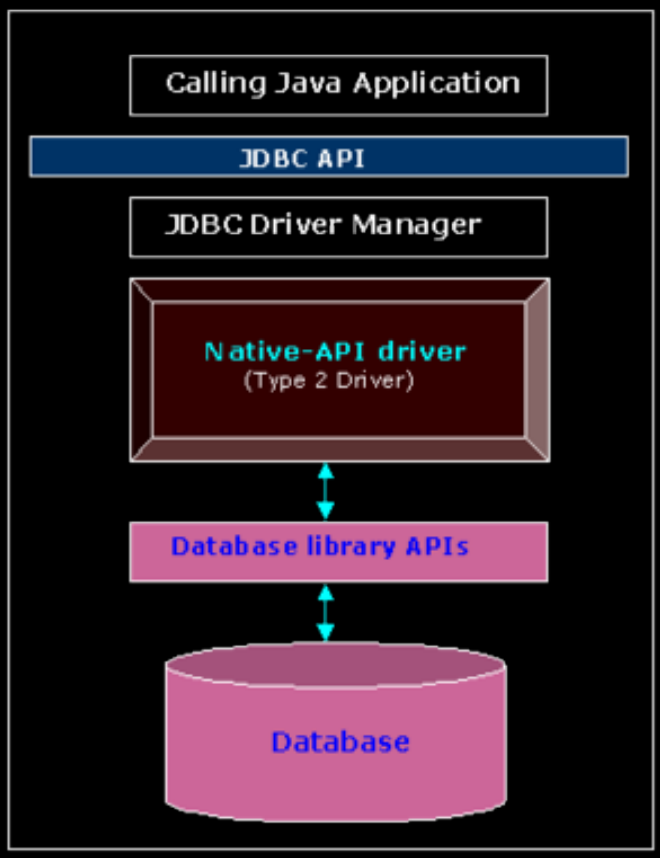
*3- y la **base** de datos.*

Tipos de conexiones JDBC

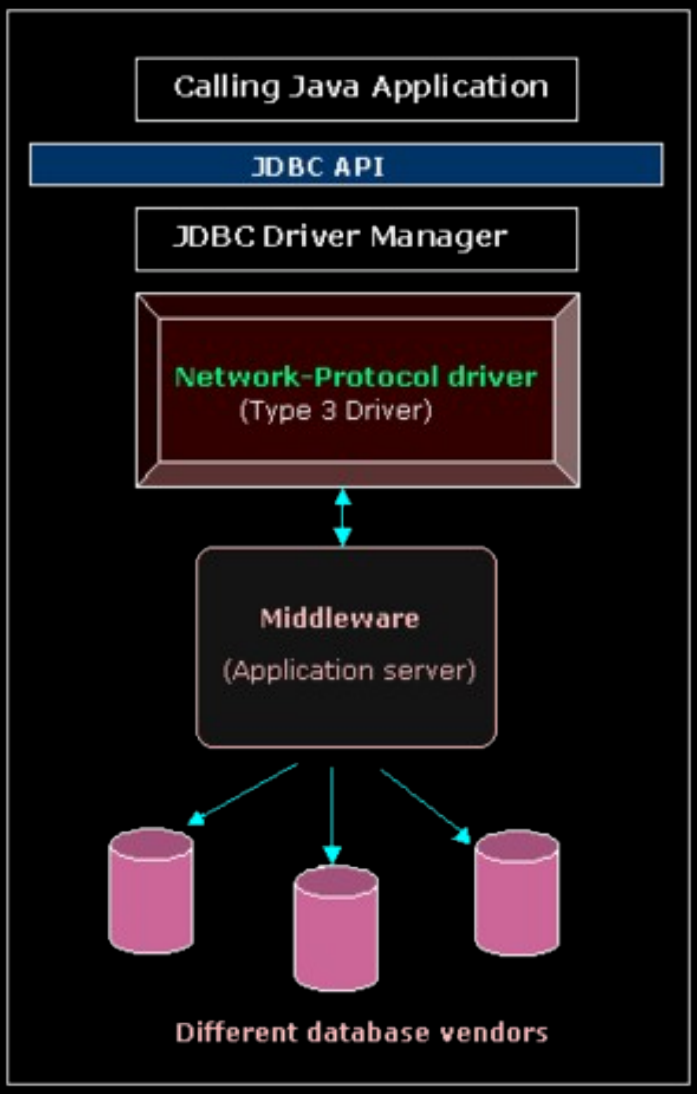
• **Driver tipo 1 JDBC-ODBC** (**Puente JDBC-ODBC**): este driver usa una **API nativa**, **traduce** las llamadas realizadas de JDBC **a ODBC**. Los **datos devueltos** por la base de datos se **traducirán a JDBC** cuando sean devueltos.



• **Driver tipo 2 JDBC Nativo (driver API-Nativo**): estos drivers están escritos una parte en **Java** y otra parte, en código **nativo**. Las **llamadas al API JDBC** son traducidas en **llamadas propias** nativas de la API **de la base de datos** relacional que tengamos. **Más rápido** que el puente JDBC-ODBC pero se necesita instalar la librería cliente de la base de datos en la máquina cliente y el driver es dependiente de la plataforma.



• **Driver tipo 3 JDBC net**: **Middleware** entre el JDBC y el SGBD. Es de **tres capas** cuyas solicitudes JDBC están siendo **traducidas** en un protocolo de red en una capa intermedia (**middleware)**. Esta capa intermedia recibirá dichas solicitudes y las enviará a la base de datos usando un driver **JDBC de tipo 1 o de tipo 2**. Es una arquitectura muy **flexible**.

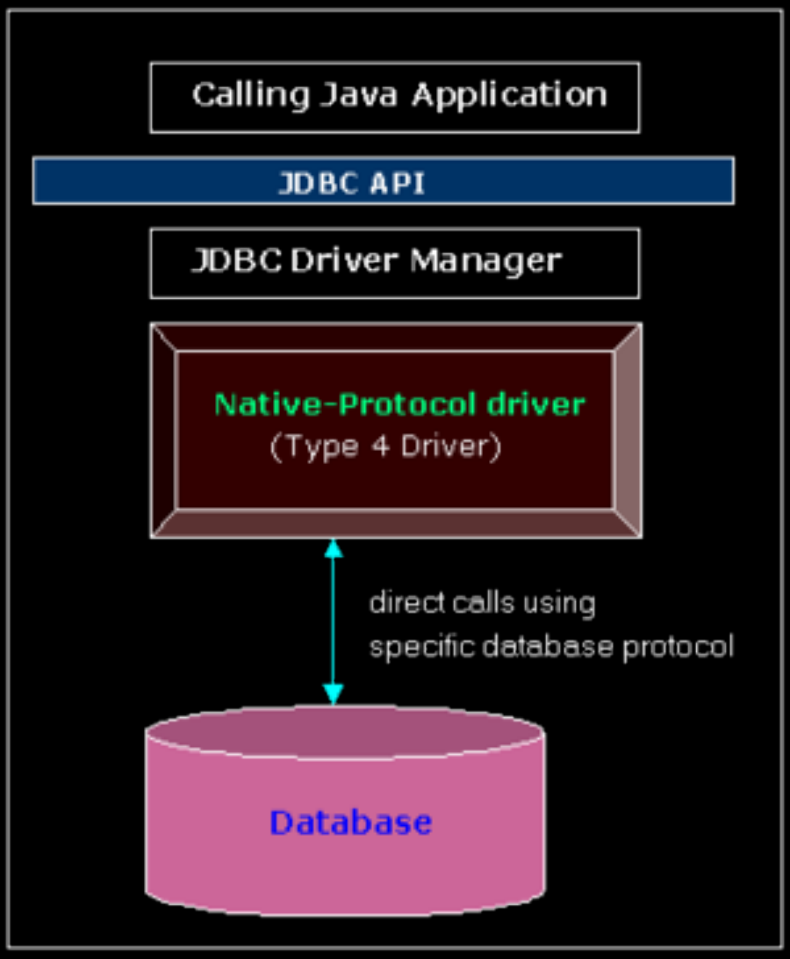


• **Driver tipo 4 protocolo nativo** (**controlador Java puro**): este tipo de driver realiza las llamadas **mediante el servidor**, usando el protocolo nativo del mismo. Estos drivers pueden desarrollarse al completo **en Java**. Si en el futuro se necesitara hacer un **cambio de base de datos**, evidentemente, habría que desarrollar **otro driver nativo** adaptado a la **nueva base** de datos relacional.

Wiki: El controlador JDBC tipo 4, también conocido como controlador Java puro directo a base de datos , es una implementación de controlador de base de datos que **convierte llamadas JDBC** directamente en un **protocolo** de base de datos **específico del proveedor**.

Escritos completamente en Java, los controladores tipo 4 son, por tanto, **independientes de la plataforma**. Se instalan **dentro de la máquina virtual Java** del cliente. Esto proporciona un mejor **rendimiento** que los controladores tipo 1 y tipo 2, ya que no tiene la sobrecarga de conversión de llamadas en ODBC o llamadas API de bases de datos. A diferencia de los controladores tipo 3, **no necesita software asociado** para funcionar.

Como el protocolo de la base de datos es específico del proveedor, el cliente JDBC requiere **controladores separados**, generalmente proporcionados por el proveedor, para conectarse a diferentes tipos de bases de datos.



Configuración de una conexión en código

Ejemplo de conexión en línea de código:

```
/**
*1- El primer paso: sería descargar el driver (suele ser ".jar") de conexión de la base
de datos que vamos a utilizar ...
2- y, a continuación, añadirlo a nuestro proyecto Java (aplicación).
El primer paso para la conexión de una base de datos externa por medio de un driver de
conexión es definir algunos literales que nos van a hacer falta, como el literal "Driver",
que hace referencia a la librería que hemos añadido a nuestro aplicativo, y la
"URL CONEXION", que hace referencia a la URL donde se alojará la información.
Estas, podemos definirlas como variables estáticas generales, ya que accederemos luego.
*/
private static final String DRIVER = "org.mysql.jdbc.Driver";
private static final String URL_CONEXION = "jdbc:mysql://localhost:3306/Pruebas";
```

Como estamos realizando una **prueba** de desarrollo, hemos introducido el código en nuestro **método main**. Normalmente se implementaría usando arquitectura DAO (**Data Access Object** o patrón de diseño en el cual una clase se encarga de las operaciones de persistencia en una tabla de la base de datos.).

```
public static void main(String args[]) throws SQLException {
    /*
    Se definen  variables de tipo String que nos van a servir para realizar la conexión con la
    base    de datos más tarde.
    Instanciamos el usuario y la contraseña de nuestra conexión y también una variable de
    tipo Connection y otra Statement.
    Normalmente se definen los literales de usuario y password en la capa DAO.
    */
    final String usuario = "user_db";
    final String password = "password_db";
    Connection dbConnection = null;
    Statement statement = null;
```

Connection es una **interfaz** que representa una **conexión directa** con una **base de datos**. El motivo de que sea una interfaz es porque tendrá **distintas implementaciones** posibles.

JDBC ofrece **distintas formas** para realizar **conexiones**. Nos centraremos en establecer la conexión con "java.sql.DriverManager", recomendada para aquellas **aplicaciones** que se hayan **desarrollado en** lenguaje **Java**.

Establecer conexión

Podremos tener instaladas tantas conexiones como queramos. Cada **conexión** y cada **base de datos** utilizará los drivers JDBC, y, a su vez, cada uno de ellos implementará la **interfaz "java.sql.Driver"** . Con el método principal **connect()**, obtendremos el objeto Connection y **estableceremos la conexión** con base de datos.

Una vez que DriverManager nos ha devuelto la conexión a base de datos, realizaremos una **consulta simple** y la almacenaremos en una variable de tipo **String** para más tarde ser ejecutada.

```
try {
        /**
        * Registramos el driver que anteriormente hemos indicado en la variable estática "DRIVER".
        * Con esta instrucción cargamos la librería "org.mysql.jdbc.Driver".
        * Con Class.forName estaremos dando de alta un driver en nuestra aplicación:
        * /
        Class.forName(DRIVER);
        /**
        * El objetivo de la clase DriverManager, realmente, es gestionar los drivers que
          poseemos en nuestra aplicación y permitir en una misma capa el acceso a todos
          y cada uno de ellos. Algo que debemos tener en cuenta es que DriverManager
          necesita que todos y cada uno de los drivers estén registrados antes de su uso.
          Las conexiones deben quedar almacenadas antes de acceder a la base de datos.
          Después de haber registrado el driver, se pueden usar los métodos estáticos
          para hacer "getConnection", usándolo directamente para establecer conexiones.
          Al método "getConnection" le pasamos por parámetro la URL de conexión previamente
          definida: usuario y contraseña.
          Nos devolverá un objeto de tipo Connection, en nuestro caso lo hemos llamado
          dbConnection. De modo que en dbConnection tendríamos la conexión.
        */
        dbConnection = DriverManager.getConnection(URL_CONEXION, usuario, password);
        /**
          Y ahora ya podemos usar la Base de Datos con sentencias, etc...
        */
        String selectTableSQL = "SELECT ID,USERNAME,PASSWORD,NOMBRE FROM Usuarios";
        /**
          Y creamos el Statement (declaración SQL) en nuestra conexión a la BDD.
          El resultado de la petición a la BDD se almacenará en un ResultSet:
          Con la variable Connection, ejecutamos el método "createStatement" y lo asignamos
          a la variable definida al principio del ejercicio de tipo Statement.
          Realizamos la consulta con el método "executeQuery"
          pasándole como parámetro la query previamente definida en la variable de tipo String.
        */
        statement = dbConnection.createStatement();
```

```java
/**
  El resultado de la query se asignará a una variable de tipo ResultSet (rs).
  La lectura del ResultSet está envuelto en un bucle “while”,
  ya que por cada fila que nos devuelva esta tabla, podremos ir dando una vuelta más
  al bucle y seguir mostrando los resultados.
  Mostraremos por pantalla tanto el ID, el USERNAME, el PASSWORD y el NOMBRE,
  que son columnas de la tabla Usuarios que hemos consultado de prueba.
*/
ResultSet rs = statement.executeQuery(selectTableSQL);
while (rs.next()) {
            String id = rs.getString("ID");
            String usr = rs.getString("USERNAME");
            String psw = rs.getString("PASSWORD");
            String nombre = rs.getString("NOMBRE");
            System.out.println("userid : " + id);
            System.out.println("usr : " + usr);
            System.out.println("psw : " + psw);
            System.out.println("nombre : " + nombre);
}
```

```
} catch (SQLException e) {
        /**
         Excepción capturada si a la hora de ejecutar el método “executeQuery”
         algo va mal en base de datos, ya sea gramaticalmente, sintácticamente, etc.
        */
        System.out.println(e.getMessage());
} catch (ClassNotFoundException e) {
        /**
         Excepción lanzada y capturada en este punto si en nuestra línea:
         “Class.forName(DRIVER)”
         el fichero del driver que le estamos indicando no encontrara la librería.
        */
        System.out.println(e.getMessage());
} finally {
        /**
         La sentencia finally se ejecutará siempre, hayamos capturado excepción o no.
         En esta, simplemente, se realizan los cierres de la clase Statement y del
         objeto Connection que, a su vez, en este punto pueden lanzar una excepción
         que será recogida y lanzada a la capa superior a través de la palabra clave
         “Throws” en la definición de nuestro método.
        */
        if (statement != null) {
                statement.close();
        }
        if (dbConnection != null) {
                dbConnection.close();
        }
}
```

Ventajas e inconvenientes del uso de conectores

Drivers tipo 1
(Puente JDBC-ODBC)

- **Ventajas**:
    - Solemos encontrarlos fácilmente, ya que se distribuyen **con el paquete** del lenguaje Java.
    - Acceso a gran cantidad de **drivers ODBC**.
- **Inconvenientes**:
    - Rendimiento: **demasiadas capas** intermedias.
    - Limitación de **funcionalidad**. (Características comunes de base de datos).
    - No funcionan bien con **applets**. **Problemas en navegadores**.

Drivers tipo 2
(driver API-Nativo)

- **Ventajas**:
    - Ofrecen **rendimiento superior** al de tipo 1, ya que son llamadas **nativas**.
- **Inconvenientes**:
    - La **librería** de la BDD, forzosamente, se inicia en la parte de **cliente**. No se pueden usar en internet.
    - **Interfaz** nativa **Java**. No movible entre plataformas.

Drivers tipo 3
(JDBC net - Middleware)

- **Ventajas**:
    - **No necesita librería del fabricante**. No es necesario llevar al cliente este aspecto.
    - Son los que mejor **rendimiento** dan en **internet**, muchas opciones de **portabilidad** y **escalabilidad**.
- **Inconvenientes**:
    - Requieren de un **código especifico** de BDD para la **capa intermedia**.

Drivers tipo 4
(controlador Java puro)

- **Ventajas**:
    - Buen **rendimiento**.
    - No **necesitan instalar un software especial** ni en la parte del servidor, ni en la parte de cliente. Drivers de fácil acceso.
- **Inconvenientes**:
    - El usuario necesitará **distinto** software de conexión (**driver**) para cada **base de datos**.

Ejemplo de clase que realice las gestiones de conexión, e instancie cualquiera de los **parámetros** que se les vayan pasando de los diferentes **drivers**.

Implementar en esa clase un método **getDBConnection**() en donde deberá pasar cuatro atributos por parámetro:

- Nombre del driver
- Url de conexión
- Usuario
- Contraseña

El método **getDBConnection()** podría ser así:

```
private Connection getDBConnection(String Driver, String url, String usuario, String password)
{
        Connection connection = null;
        try {
                Class.forName(Driver);
                connection= DriverManager.getConnection(url, usuario, password);
        } catch (ClassNotFoundException e) {
                e.printStackTrace();
        } catch (SQLException e) {
                e.printStackTrace();
        }
        return connection;
}
```

Se podrán **instanciar tantas conexiones** con bases de datos **como queramos.**

Recuerda:

Carga del driver

```
private static final String DRIVER = “nombre_del_Driver”;
private static final String URL_CONEXION = “url_de_la_base_De_datos”;
```

Almacenamiento de credenciales:

```
final String usuario = “usuario”;
final String password = “contraseña”;
```

Abrir la conexión:

```
Class.forName(DRIVER);
dbConnection = DriverManager.getConnection(URL_CONEXION, usuario, password);
```

En un **método, podemos devolver la conexión** ya establecida con nuestra URL de conexión, usuario y contraseña configurado y preparado para trabajar con la base de datos:

```
Class.forName(DRIVER);
return DriverManager.getConnection(URL_CONEXION, usuario, password);
```

http://decodigo.com/java-conexion-a-base-de-datos-con-jdbc

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

BBDD embebidas e independientes

Gestores de bases de datos embebidos e independientes

Diferentes **variantes, opciones o posibilidades** **que tenemos de almacenamiento** de datos en sistemas de bases de datos relacionales:

• **Bases de datos en memoria**: Almacena toda la información de la misma en **memoria principal del sistema (RAM)**. **No** se almacenará ningún dato **en disco**. Es muy rápida ya que el acceso a la memoria RAM es mucho más rápido que que el acceso a disco.

• **Bases de datos embebidas**: Es **parte de la aplicación** que se ha desarrollado. Accederemos por medio de los **JDBC driver**. El **motor de la base de datos** corre con el **mismo motor de la aplicación** Java (JVM-Java Virtual Machine) mientras la aplicación está en ejecución. **Desventaja:** la **cohesión** y la **dificultad de mantenimiento** que podemos encontrar en estos casos. Encaja perfectamente con la idea de tener un **repositorio** para **persistir las transacciones** sin ningún tipo de intervención por parte del usuario.

• **Bases de datos independientes SGBD** (Sistemas de Gestión de Bases de Datos relacionales): Tenemos un **gestor o administrador** de base de datos dedicado a **resolver** ciertos problemas de **mantenimiento**. Las bases de datos **cliente/servidor** son **independientes** y las más **pesadas y potentes** (cantidad de **procesos adicionales**).

Gestores de base de datos embebidos

• **HyperSQL**: se ajusta a la versión estándar **SQL 2011** y a la especificación **JDBC 4**. Además, **soporta** cada característica clásica de las **bases de datos modernas** de tipo **relacionales**. Se puede ejecutar tanto en modo **embebido** como en modo cliente/**servidor**. La base de datos es bastante **estable**, es de una **confianza** considerable, como los proyectos desarrollados de código abierto "OpenOffice" y "LibreOffice".

Desarrollado en **Java**, corre sin problema en nuestra **JVM**, facilitándonos una **interfaz JDBC** para el **acceso** a datos. El paquete principal que nos descargamos contiene un fichero .**jar** llamado "**hsqldb.jar**" dentro del directorio **/lib**, que contiene el componente requerido: el motor de la **base de datos HyperSQL RDBMS** y el **driver JDBC** embebido en la aplicación Java.

A partir de la **versión 2.3.X** en adelante soporta el mecanismo **MVCC (Multi version concurrency control)**:

> Este concepto sienta su base cuando **dos o más usuarios** de bases de datos **acceden** a la misma. Con el concepto MVCC **evitamos el tema de los bloqueos**, ya que lo que se estaría **viendo** con nuestro **acceso** es algo parecido a una **imagen** de la base de datos, por tanto, **no interferiríamos** en las transacciones.

• **ObjectDB**: también contiene los dos modos, **embebido** y modo cliente/**servidor**. No es exactamente una base de datos relacional, sino una **base de datos orientada a objetos** con soporte para la especificación **JPA2 (Java Persistence API)**. Como resultado, el uso de una **capa de abstracción**, como la de **Hibernate**, tiene un **mejor rendimiento** que cualquier otra base de datos. Debido a su compatibilidad por defecto con JPA, **se podría eliminar la capa ORM** directamente, si es que nuestro aplicativo tuviera una para aprovechar el rendimiento. Un ORM (**Object Relational Mapping**) es un **modelo de programación** cuya misión es **transformar las tablas de una base de datos** de forma que las tareas básicas, que realizan los programadores, estén simplificadas.

• **Java DB y Apache Derby**: Son muy similares, de hecho, **Java DB** está construida **sobre el motor** de la base de datos **Derby**. Esta está escrita por completo en el lenguaje **Java** y puede ser incluida, **fácilmente**, en cualquier **aplicación Java**.

Derby **soporta** todas las funcionalidades estándar de una base de datos **relacional**. Puede ser desplegada en el modo simple **embebido** o también en el modo cliente/**servidor** como sus competidores. Para **embeber** nuestra base de datos Derby en nuestra aplicación Java, simplemente, hay que incluir en nuestro proyecto Java el fichero "**derby.jar**" del directorio **/lib**. Este fichero contiene tanto el propio **motor** de la base de datos como los **conectores** necesarios para realizar la **conexión con el driver JDBC** desde código.

- **H2 Database**: las principales características de la base de datos H2 son:
    - Es muy **rápida**, de **código abierto**, **JDBC** API.
    - Contiene modo cliente/**servidor**, modo **embebido** y modo **desplegable en memoria**.
    - Se puede usar perfectamente para **aplicaciones web**.
    - Muy manejable y **transportable**, el fichero .jar ocupa **2MB** de espacio total.
    - Soporta **MVCC** (Multiversion concurrency control).
    - Se puede usar **Postgress OBDC** driver. PostgreSQL es un **sistema o motor de bases de datos** compatible con los **servicios de OVHcloud** y la mayoría de las herramientas más populares del mercado. Es compatible con diversos **modelos de datos para crear aplicaciones orientadas a objetos**, potentes y escalables.

"Spring boot database" es de ayuda para crear proyectos (incluidos los web) con SGBD embebidos.

Comparativa de gestores de bases de datos embebidos

	H2	DERBY	HYPERSQL	MYSQL	POSTGRESS
Java Puro	SI	SI	SI	NO	NO
Modo memoria	SI	SI	SI	NO	NO
Base de datos encriptada	SI	SI	SI	NO	NO
Driver ODBC	SI	NO	NO	SI	SI
Búsqueda texto completo	SI	NO	NO	SI	SI
MVCC	SI		+versión 2.3.X	SI	SI
Espacio (embebido)	~2 MB	~3 MB	~1.5 MB		
Espacio (cliente)	~500 KB	~600 KB	~1.5 MB	~1 MB	~700 KB

H2: buena opción para agregar una base de datos **embebida**, en memoria, o de tipo cliente/servidor.

Instalación de base de datos H2 en aplicación web Java usando Spring Boot

Spring es un framework muy conocido. Spring Boot es un **módulo dentro de Spring:** hace **fácil** ciertas partes del proceso que por defecto son más complejas.

En un proyecto **web Java** con **Maven** existen varias etapas como seleccionar las **dependencias** de las librerías que vamos a usar, construir el aplicativo o desplegarlo a un servidor. Spring Boot facilita al máximo la **selección de dependencias** el **despliegue de la aplicación**.

Nos será de gran utilidad añadir una base de datos que facilite el almacenamiento de información para nuestra aplicación web.

En https://start.spring.io/ se nos ofrece un entorno sencillo para crear el arquetipo de nuestra aplicación con distintas variables:

• **Opción project**, elegiremos **maven**. Ya que nuestro ejemplo será implementado y compilado con **dependencias maven**.

• **Opción Spring boot**, dejaremos la release version que venga **por defecto**, ya que será **la más estable** por el momento.

• **Opción project metadata**:

- **Group**: podemos dejarlo tal como está, ya que vamos a realizar una **prueba**.
- **Artifact**: igual que el anterior.
- **Name**: la misma idea, podemos dejarlo como aplicación **Demo**.
- **Description**: incluiremos una descripción, por ejemplo, “Hola mundo con base de datos embebida H2”.
- **Package name**: podemos dejar el nombre que nos asigna **automáticamente**.
- **Empaquetado**: elegiremos .**jar**.
- **JDK Java**: podemos trabajar bien con el **JDK 8**.

• En **Dependencies** incluiremos nuestra **base de datos H2 embebida**.

Nos **permite incluir la mayoría de los framework y utilidades** que rodean el lenguaje por medio de **dependencias** que serán incluidas **automáticamente**.

Incluiremos nuestra base de datos,

Hemos **añadido la base de datos H2** y el módulo "**Spring Web**" (vamos a simular la funcionalidad de una **aplicación web** cliente/servidor). Tambien incluiremos **Spring Data JPA**. Hacemos clic en "GENERATE" y automáticamente se nos descargará un fichero con extensión .zip o .rar, y tendremos un proyecto web java con todas las características.

Descomprimimos el directorio del proyecto, y lo abrimos **desde el IDE** que usemos frecuentemente (Intellij, Eclipse, Netbeans...). En el archivo "**pom.xml**" Spring Boot nos ha añadido por defecto algunas de las **dependencias** que nosotros le indicamos anteriormente en la interfaz:

```
<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>
	<scope>runtime</scope>
</dependency>
```

Las **librerías** de la **base de datos H2** han sido **añadidas** y perfectamente **integradas** por Spring Boot.

Nos genera un **fichero** llamado "**Application.properties**" (en **src/main/resources**/ de la raiz del proyecto) donde tenemos algunos aspectos de **configuración básica** de nuestra **base de datos H2**:

Lo siguiente no es necesario agragarlo al archivo para que funcione nuestra aplicación web, aunque debemos de entender qué hace cada configuración:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Con la instrucción `spring.datasource.url=jdbc:h2:mem:testdb`, podemos ver como Spring Boot nos ha configurado nuestra base de datos para que el almacenamiento de la misma se realice en **memoria del sistema**. Evidentemente, como es volátil, la información estará visible mientras la aplicación esté corriendo, pero no hay ningún tipo de problema, porque si queremos cambiar a tipo **embebida con almacenamiento en disco** ejecutaremos algo así:

```
spring.datasource.url=jdbc:h2:file:~/data/demo
```

Con esto se guardaría en disco los cambios en nuestra base de datos. Nuestra **base de datos quedará almacenada en la ruta** que se indica.

Con **spring.datasource.driverClassName** indicaremos el nombre del driver en el fichero .properties de Spring Boot, y a continuación, deberíamos indicar el nombre relativo del driver.

Con **spring.datasource.username=sa** y **spring.datasource.password=password** agregamos las credenciales para la conexión a la base de datos. Estas las podremos cambiar desde este archivos cuando lo necesitemos y surtirán efecto en ese momento.

Iniciar la BBDD

Nuestra base de datos está montada y preparada para funcionar. Sería ideal montar una buena **capa DAO** (patrón **Data Access Object**) de acceso a la capa de datos en nuestro aplicativo, ya que tendremos todas las opciones de persistencia y el potencial que nos ofrece **JPA** (JPA es la propuesta estándar que ofrece Java para implementar un Framework Object Relational Mapping (ORM), que permite **interactuar con la base de datos por medio de objetos**). Nosotros, para hacer **un test** de la misma, vamos a indicar otros ficheros de configuración.

En la ruta "**src/main/resources**" añadiremos un fichero llamado "**data.sql**". La próxima vez que arranque nuestra aplicación, **Spring Boot** cogerá dicho fichero y **lo ejecutará** al comienzo de la misma, por lo tanto, este será un buen momento para la **creación de una tabla**, o borrado de alguna que estuviera con anterioridad; en definitiva, realizar **operaciones de limpieza y preparado de datos**:

En principio para que nuestra aplicación funcione tampoco es necesario agregar este archivo y añadir estas sentencias SQL, pero es conveniente saber utilizarlo:

```
DROP TABLE IF EXISTS RESERVATION;
CREATE TABLE RESERVATION (
    id INT AUTO_INCREMENT PRIMARY KEY,
    reservation_date TIMESTAMP NULL DEFAULT NULL,
    name VARCHAR(250) NULL,
    status VARCHAR(25) NOT NULL DEFAULT 'FREE'
    );
```

Otro dato **importante** a tener en cuenta es cómo accedemos a la **consola de nuestra base de datos** o a nuestro panel de gestión de la base de datos. Añadimos en nuestro fichero "**application.properties**" la siguiente línea:

```
spring.h2.console.enabled=true
```

Cuando nuestra aplicación esté ejecutándose, nos dirigiremos a nuestro navegador e introduciremos la siguiente URL:

http://localhost:8080/h2-console

De esta forma, accederemos a nuestro **panel de control de la base de datos**, donde podremos realizar distintas operaciones.

Dos maneras de agregar **base de datos Apache Derby** a nuestra aplicación de manera **embebida**:

1. Descargar el .**jar** de la página oficial de Apache Derby:

https://db.apache.org/derby/derby_downloads. html#For+Java+8+and+Higher.

Y añadiremos este fichero a nuestro aplicativo, el cual contiene el motor de la base de datos y el conector.

2. Directamente montar nuestro proyecto con **Spring Boot:** En spring.io elegiremos la **dependencia de Derby DB (Apache Derby Database sql)** para la creación de nuestro proyecto atendido por Spring Boot.

Gestores de base de datos independientes

Una base de datos independiente no puede ejecutarse bajo la misma máquina virtual de Java que usa nuestra aplicación en ejecución. Puede ser instalada **en local**, pero **con fines de prueba o aprendizaje** en la mayoría de casos. **Lo común** es tener una **base de datos independiente** separada e instalada **en otra máquina**.

Consumirá siempre más recursos que una embebida. Al estar en una **máquina aislada** para dicha base de datos, **aprovechará mejor los recursos** que tiene disponibles. Se preparan en dicha máquina separada y aislada para **volcar todo el potencial** que ofrece con todos sus **procedimientos y operativos**.

Las **más conocidas** a nivel comercial y de mayor potencia y extensión:

- **SQL Server DataBase.**
- **Oracle.**
- **Postgres SQL.**
- **MySql.**

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Sentencias SQL
y
Transacciones

Aprenderemos las **sentencias principales** de **definición** de datos, de **manipulación** de datos.

Veremos ejemplos de cómo **obtener información** de nuestra fuente de datos, así como realizar consultas.

Recordaremos qué son las transacciones SQL.

Sentencias de definición de datos

Creación de base de datos

Estas sentencias son usadas en **base de datos MySQL**.

• **ALTER DATABASE**: Cambiar **características generales** de una **base de datos**. Estas características son almacenadas en un **archivo** que suele ser "**db.opt**". Para usar este comando se necesita el **permiso ALTER**.

• **ALTER TABLE**: Modificar la **estructura** de una **tabla** que previamente hemos creado. Se pueden realizar numerosas acciones como **agregar columnas**, **borrarlas**, crear **índices**, borrarlos, cambiar la **tipología** de ciertas **columnas**, **renombrar** las columnas, también modificar la **descripción** de la **tabla** y la **tipología** de la misma.

• **CREATE DATABASE**: Creación de una **nueva base de datos**, para ello es necesario disponer del **permiso CREATE**.

"**create_specification**": se **establecerán** las distintas **características** de la base de datos. También están almacenadas en el fichero con **extensión .opt** en la raíz de la base de datos. "**character_set**": se especificará el **set de caracteres por defecto** de dicha base de datos que se está creando.

• **CREATE INDEX**: En muchas de las bases de datos, este comando **se traduce** a un comando **ALTER TABLE** para la creación de índices. Normalmente, con la creación de la tabla, se agregan todos los **índices**, pero con este comando podemos **agregarlos manualmente después de haber creado una tabla**. Para el tipo de columnas numeradas (columna1, columna2, etc.), se crea un índice de columnas múltiples. Los **índices** se forman **al unir los valores de las columnas**.

Definición de datos

*Tal y como sabemos, el comando ALTER TABLE nos modifica una tabla de base de datos. Un dato a tener en cuenta es que las **columnas están numeradas** desde el número 1 en adelante.*

*Podemos utilizar el **parámetro POSICIÓN** si estamos añadiendo una **nueva columna** en la tabla, para indicarle en qué **posición** de la misma **queremos colocarla**.*

Creación y eliminación de tabla

• **CREATE TABLE**: **Creación** de una tabla con el nombre definido. Es evidente que también necesitaremos el **permiso CREATE** para la tabla. Existen algunas **restricciones** a la hora de establecer un nombre a la tabla definida. Ocurrirá un **error** si la tabla que estamos creando **existe previamente** en la base de datos que estamos trabajando. Se puede usar "**TEMPORARY**" y será solo visible **mientras tengamos activa la conexión** con la que estamos trabajando, después, dicha tabla no existirá.

También podremos usar las **claves** "**IF NOT EXISTS**" para asegurarnos de que cierta tabla no existe.

• **DROP DATABASE**: Realizar un **borrado permanente** de todas las **tablas** de nuestra base de datos y borrar, así, dicha **base de datos**. Es un comando muy peligroso; por ello tendremos que tener habilitado el **permiso de DROP**. Si la base de datos que estamos borrando está **enlazada simbólicamente**, se **borrarán ambos objetos**.

• **DROP INDEX**: Añadiendo el **nombre del índice** y la **tabla** especificada, se ejecutará un **ALTER TABLE** justo para **borrar el índice** que estamos indicando.

• **DROP TABLE**: Borrar **una o más tablas** en nuestra base de datos. El **permiso DROP** debe estar habilitado para nuestro usuario de la base de datos.

Es otro comando que deberá de ser ejecutado con mucha precaución, ya que toda la información de la tabla que estamos indicando será eliminada. También podremos usar la clave "**IF EXISTS**" para evitar el error de cuando la tabla no exista. Este comando DROP TABLE realizará **commit automáticamente** al ser ejecutado.

• **RENAME TABLE**: Renombrar **una o más tablas**. Una vez se está ejecutando el renombrado, la tabla quedará temporalmente **bloqueada hasta finalizar la transacción**.

Sentencias de manipulación de datos

Inserción

• **DELETE**:

Eliminamos las filas de “tbl_name” que validan la condición expresada en “where_definition”.

Este comando nos **devolverá el número de registros** eliminados. **Si no** establecemos clausula “**where**”, se **eliminarán todas** las filas de la tabla.

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
[WHERE where_definition]
[ORDER BY ...]
[LIMIT row_count]
```

• **DO**:

Ejecutaremos expresiones sin obtener ningún **resultado**. Realmente, es una forma de realizar **SELECT** (expresión), pero con la ventaja de que **es más rápido** cuando no es de interés el resultado.

```
DO expr [, expr] ...
```

• **HANDLER**: Accederemos directamente a las **distintas interfaces** del **motor de la tabla**, tendremos **comandos complementarios** como OPEN, READ o CLOSE para **leer ciertos datos** de dicha tabla.

• **INSERT**:

Agregar **nuevos registros** en una tabla previamente definida. En relación a la forma **INSERT-SET** o **INSERT-VALUES**, se insertarán **valores basados en las columnas** de la tabla. También podremos encontrar **INSERT-SELECT** cogiendo registros de **otra/s tabla/s**.

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] tbl_name [(col_name,...)]
VALUES ({expr | DEFAULT},...),(...),...
[ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

• **LOAD DATA INFILE**: L**eer los registros**.

Edición

• **REPLACE**:

Misma función que INSERT con una excepción: se **sobrescribirán** los registros para los índices de tipo PRIMARY KEY o UNIQUE teniendo en cuenta que el **registro anterior se borra** antes de agregar el nuevo registro. Solo tendrá sentido, evidentemente, si la tabla contiene este tipo de índices.

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] tbl_name [(col_name,...)]
VALUES ({expr | DEFAULT},...),(...),...
```

Ejemplo:

```
CREATE TABLE test (
        id INT UNSIGNED NOT NULL AUTO_INCREMENT,
        data VARCHAR(64) DEFAULT NULL,
        ts TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
        CURRENT_TIMESTAMP,
        PRIMARY KEY (id)
);

mysql> REPLACE INTO test VALUES (1, 'Old', '2014-08-20 18:47:00');

mysql> REPLACE INTO test VALUES (1, 'New', '2014-08-20 18:47:42');

mysql> SELECT * FROM test;
+----+------+---------------------+
| id | data | ts |
+----+------+---------------------+
| 1 | New | 2014-08-20 18:47:42 |
+----+------+---------------------+
```

• **SELECT**: Realizar **consultas** de registros de una o más tablas. Podremos realizar consultas **simples, complejas o subconsultas**. En el siguiente punto, dedicaremos todo un punto a estudiar las consultas y subconsultas, veremos su sintaxis y más detalles.

• **TRUNCATE**:

Sintaxis de TRUNCATE:

```
TRUNCATE TABLE tbl_name;
```

Eliminar completamente una tabla. Es equivalente a un **DELETE**, pero con ligeras diferencias. Dependiendo del motor de base de datos, en algunos se realiza un DELETE tal como lo hemos estudiado, se borran todos los registros de esa tabla; y para otros motores de base de datos, se elimina el objeto completo de la tabla y se vuelve a crear. Hay que tener en cuenta que este tipo de operaciones **no son transaccionales** (que estudiaremos más adelante), lo que significa que nos dará un **error si dicha tabla** está **ocupada** en ese momento.

• **UPDATE**:

Sintaxis de UPDATE:

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
SET col_name = expr1 [, col_name = expr2 ...]
[WHERE where_definition]
[ORDER BY ...]
[LIMIT row_count]
```

Actualizará la información de las columnas que le indiquemos en la sección **SET** con **información nueva**. Con la cláusula **WHERE** indicaremos qué registros deben actualizarse, y **si no** existe esta cláusula, se **modificarán todos**.

Ejemplo UPDATE:

```
UPDATE empleados
SET sueldo_bruto = '50000', prima_objetivos = '3000'
WHERE sueldo_bruto < 45000 AND sueldo_bruto > 40000
ORDER BY antiguedad DESC
LIMIT 50
```

- ➔ **Establecer** el sueldo bruto anual a 50.000 dólares y la prima de objetivos a 3.000.
- ➔ Esto será **efectivo a** los empleados que cobren entre 40.000 y 45.000 dólares.
- ➔ Se actualizarán los **primeros 50 empleados** (**LIMIT 50**) ordenados de **mayor a menor** antigüedad en la empresa.

Consultas

SELECT

Una consulta es realizar una **pregunta** a base de datos con una serie de criterios, y esta ejecutará una respuesta a dicha pregunta. Podremos consultar **una tabla o más de una**.

Existen distintas **cláusulas vinculadas a SELECT**: podemos encontrar cláusulas de tipo **HAVING**, también podemos encontrar **ORDER BY**, **UNION**... la sentencia SELECT se puede combinar de diversas formas.

Cuando realizamos una consulta con SELECT, lo que **obtenemos** es una **tabla ficticia**, una serie de **resultados que se relacionan** acorde a nuestros requisitos en la consulta. Esta tabla de la que hablamos, evidentemente, no persiste en disco ni nada por el estilo, se mantiene **en memoria mientras** la estamos **usando**.

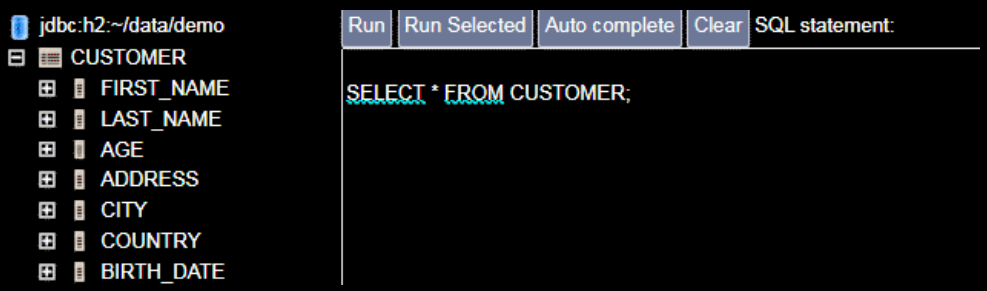
A continuación, veremos algunos de los **puntos principales** de la sintaxis de la sentencia **SELECT**:

• **SELECT**: Junto a la palabra clave "SELECT", podremos añadir [ALL/DISTINC], también asterisco [*] seguido de un listado de columnas [columnas], incluso podremos realizar un **alias** con la palabra clave **AS [nombre del alias]**. Con esta sintaxis podremos **seleccionar ciertas columnas** que van a ser mostradas junto con el orden.

Con **ALL** mostrará filas **duplicadas**.

Con **DISTINCT** incluirá en el resultado solo filas **únicas**.

• **FROM**: Usaremos la palabra clave "FROM" y a continuación el **nombre de** la **tabla/s** o **vista/s**. Hasta aquí, con los conceptos básicos que tenemos, podríamos ejecutar perfectamente una consulta básica pidiendo **todas las filas con todos sus campos** de una supuesta **tabla "CUSTOMER"**:



• **WHERE**: Especificaremos algunas **condiciones de filtro**. Usaremos dicha cláusula cuando **no queramos obtener el contenido total** de la tabla o tablas que estemos consultando. A continuación, veremos algunas de las condiciones que podremos agregar a nuestras consultas.

WHERE

Vamos a aprender a introducir algunas **condiciones** y **expresiones lógicas** en ella, de tal forma, que harán de **filtro** a la hora de preguntar a nuestra base de información y obtención de resultados.

Algunas **condiciones**:

• **Operadores**: podemos encontrar mayor que ("**>**"), mayor o igual que ("**>=**"), menor que ("**<**"), igual ("**=**"), distinto que podemos expresarlo: ("**<>**") o también: ("**!=**").

• **Nulos**: podremos añadir **IS NULL** o **IS NOT NULL** si lo que queremos es comprobar que el valor de cierta columna sea nulo o no. Un valor es nulo cuando **no existe valor**, un registro **en blanco no sería nulo**.

• **LIKE**: realiza una **comparación con los registros**. Se usarán caracteres especiales. Estos caracteres son "_" y también "%".

"**%**" indicamos que puede ir **cualquier cadena de caracteres** en esa posición.

"_" sería el mismo concepto, pero, en este caso, **solo con un carácter**.

Ejemplos:

• **WHERE** APELLIDO **LIKE** 'C%': mostraremos resultados que coincidan donde el apellido empiece por "C" seguido que cualquier cantidad de caracteres.

• **WHERE** APELLIDO **LIKE** 'A_': en este caso, el apellido empezaría por la letra "A" y luego le seguiría 1 solo carácter.

• **BETWEEN**: **Rango de valores.**

Ejemplo:

• **WHERE** EDAD **BETWEEN** 3 AND 7: mostraremos aquellas edades que estén entre 3 y 7, incluyendo dichos valores.

• **IN ()**: Mostrar una serie de resultados cuyos **valores coincidan** con los especificados en la **clave**. Ejemplo:

• **WHERE** EDAD **IN** (3, 4, 7, 8): en este caso mostraremos, por ejemplo, los usuarios cuya **edad coincida con las especificadas** en la clave IN.

Ejemplos con WHERE

Podremos **combinarlos con operadores lógicos OR** (o), **AND** (y) y **NOT** (negativa), y ayudarnos de los **paréntesis** para establecer **prioridades** entre ellos.

• **ORDER BY**: **Después del WHERE.** Como su nombre indica, la usaremos para establecer un orden a la hora de mostrar resultados según el campo o campos por los que queramos ordenar.

Podremos ayudarnos de **ASC** o **DESC** si lo que deseamos es mostrar los resultados en orden **ascendente** en el caso ASC (de menor a mayor) o mostrarlos en orden **descendente** en el caso de DESC (de mayor a menor).

Ejemplos de consultas.

En esta consulta, estamos mostrando **todas las columnas de la tabla "CUSTOMER"**:

```
SELECT * FROM CUSTOMER;
```

FIRST_NAME	LAST_NAME	AGE	ADDRESS	CITY	COUNTRY	BIRTH_DATE
juan	Lopez	10	avd jaen	Badajoz	Spain	1980-06-04 00:00:00
pepe	Tejada	20	avd 1	Sevilla	Spain	1980-06-04 00:00:00
victor	Fernandez	22	avd 2	Jaen	Spain	1960-06-04 00:00:00
jose	Assensio	40	avd 3	Huesca	Spain	1930-06-04 00:00:00
Leo	Perez	30	avd 4	Orense	Spain	2000-06-04 00:00:00
Patricia	Aranda	11	avd hoho	Barceloona	Spain	1988-06-04 00:00:00
David	Lopez	105	avd peugeot	Madrid	Spain	1966-06-04 00:00:00
Enrique	Ferreras	70	avd jujer	Malaga	Spain	1980-06-04 00:00:00
Seluis	Guzman	45	avd Madrid	Valencia	Spain	1963-06-04 00:00:00

(9 rows, 8 ms)

Consulta seleccionando todos los campos de la tabla CUSTOMER, **filtrando** por aquellos cuyo campo "LAST_NAME" **empiece por** "L" **o por** "F" y, además, **ordenados de mayor a menor** por el campo "AGE":

```
SELECT *
FROM CUSTOMER
WHERE (LAST_NAME LIKE 'L%') OR (LAST_NAME LIKE 'F%')
ORDER BY AGE DESC;
```

FIRST_NAME	LAST_NAME	AGE	ADDRESS	CITY	COUNTRY	BIRTH_DATE
David	Lopez	105	avd peugeot	Madrid	Spain	1966-06-04 00:00:00
Enrique	Ferreras	70	avd jujer	Malaga	Spain	1980-06-04 00:00:00
victor	Fernandez	22	avd 2	Jaen	Spain	1960-06-04 00:00:00
juan	Lopez	10	avd jaen	Badajoz	Spain	1980-06-04 00:00:00

(4 rows, 7 ms)

Transacciones

Son unidades o **conjuntos de acciones** que se realizan en **serie y de forma ordenada** en el sistema gestor de base de datos.

Los **objetivos**:

- Proporcionar **consistencia** en la base de datos realizando **secuencias de alta fiabilidad**, de tal forma que se pueda **volver a estados anteriores** fácilmente.
- Ofrecer **aislamiento** cuando más de un aplicativo está **accediendo** a los datos **simultáneamente**.

Comandos de control que se realizan para la ejecución de transacciones en SQL:

- **Commit**: con este comando se **persistirán** los cambios en base de datos.
- **Rollback**: **desharemos** los cambios que se hubieran ejecutado hasta el momento y se **abandonará la transacción**.
- **Savepoint**: puntos donde se podrá almacenar y, en caso de **rollback**, se podrá **volver a dicho punto** de control.

La interfaz Statement

Esta interfaz es usada **cuando se realiza una conexión con un driver** de una base de datos.

Es la **encargada de ejecutar sentencias** en nuestra aplicación y **recoger los resultados** para manipularlos más tarde.

Una vez se crea el **objeto Statement**, disponemos de un lugar adecuado para **realizar consultas SQL**. Podremos usar diferentes **métodos** para ello:

- **executeQuery (String)**: **Realiza sentencias SELECT**, siendo consultas que como resultado, este método nos **devolverá un objeto ResulSet** con toda la información resultante.
- **executeUpdate (String)**: Realiza **sentencias de manipulación** de datos, ya sean **INSERT**, **DELETE**, **UPDATE**, etc. Una vez ejecutada la sentencia que le indiquemos, como String, nos **devolverá un entero** que contiene la **cantidad de filas** que han sido **afectadas** en la operación.
- **execute (String)**: **Ejecuta cualquier acción query o update**.

  Devolverá **true** si **devuelve un Resulset**, y para acceder a él, tendremos que ejecutar el método **getResulSet ()**.

  Devolverá **false**, si lo que estamos ejecutando, por ejemplo, es un **UPDATE.** En ese caso, si queremos **saber las filas afectadas** consultaríamos el método **getUpdateCount ()**.

ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

El mapeo objeto relacional
ORM
Hibernate

Concepto de mapeo objeto relacional

Java es un lenguaje de programación **orientado a objetos** el cual puede ser **representado en un gráfico de objetos**; mientras que una **base de datos relacional** se representa en un formato tabular usando **filas y columnas**.

Cuando se trata de **grabar un objeto en base de datos**, existen algunas diferencias obvias entre estos dos sistemas; por lo tanto, para solventar este tipo de problemas o diferencias de datos a almacenar, tenemos la **ayuda** del **Mapeo Objeto Relacional**.

Un **ORM** es un **framework** que facilita el almacenamiento de los **objetos** en una **base de datos relacional**.

En una base de datos, el **nivel de elemento más alto** es la **tabla**, dividida a su vez en **filas y columnas**. Una **columna** contiene **valores en un tipo** determinado, y una **fila** contendrá un **conjunto de información** de una tabla determinada.

Un **objeto plano** es equivalente a **una fila en una base de datos relacional**, y básicamente esto es lo que se mapea en un sentido y en otro.

> ***ORM Hibernate Espejo***
>
> *En un ORM como Hibernate son una **réplica a nivel de definición** **de datos:***
>
> - *Lo que tenemos en el lado de **base de datos relacional**,*
> - *Lo que tenemos en el lado de **aplicación.***
>
> *Deberán de **definirse las entidades tal y como** están definidas en la **base de datos** real para que no haya **ningún tipo de discordancia** y se puedan efectuar las operaciones sin ningún tipo de incongruencia.*

Flujo del mapeo objeto relacional

Aplicación Java → Objeto Java → ORM (framework) → Base de Datos

*(objetos en **entidades fáciles de persistir**)*

Ventajas e inconvenientes del mapeo objeto relacional

Java posee una **API de conexión a base de datos**: los drivers de conexión (**JDBC**) para acceder a la base de datos, lo que nos facilita la forma de **consultar dicha información** en base de datos.

Escribiríamos ciertas consultas SQL nativas y obtendríamos un Resulset con el resultado de nuestra Query. Para hacer esto, el desarrollador debe conocer la base de datos a fondo, y saber qué tipo de **relaciones existen entre tablas**, así como el **nombre exacto de las columnas**, **Constrains**, etc.

Si consiguiéramos **realizar las mismas operaciones** con los datos **desde la parte de Java**, la perspectiva cambiaría totalmente. De esta forma tendríamos **abstracción, herencia, composición, identidad** y **muchas características** más en la parte de la aplicación Java que se podrían conseguir igualmente por medio de un framework (algunas de las razones por las que se usan los **ORM**).

Existen **distintos tipos de base de datos**, cada una con diferentes tipos de funciones y tipología de datos definidos. Cuando usamos la conexión **JDBC** debemos tener **en cuenta** este tipo de **diferencias**.

Algunas de las **ventajas de usar un ORM**:

- Mejora en la **eficiencia** del desarrollo.
- Desarrollo más **orientado a objetos**. Mejora del **desarrollo orientado a objetos.**
- Mejora de la **manejabilidad** de los **objetos** de la aplicación.
- Mejora a la hora de realizar diferentes **consultas y procedimientos**.
- Facilidad para **introducir nuevas funciones** (cacheo de información…).
- Mejora de la **manejabilidad**.

Algunas de los **inconvenientes**:

- El mapeado automático de las bases de datos **consumen muchos recursos** de sistema.
- Aumento de la memoria de la carga de recursos al introducir esta capa intermedia del ORM.
- La **sintaxis de los ORM** a veces puede **complicarse** si realizamos **consultas muy complejas** mediante las que crucemos varias **tablas** y con diversas **condiciones**.

Fases de mapeo objeto relacional
Arquitectura funcional de un framework ORM

Tres fases de funcionamiento:

- **Fase 1 (dentro de la app)**: Estructura...

  Nos centramos en los **datos del objeto**. Esta fase contiene los **POJO** (**Plain Old Java Object**), las clases simples de Java, las clases de implementación, clases e interfaces con su correspondiente **capa de negocio** de cada aplicativo (a esta capa la podemos llamar **capa servicio** y en ella también encontraremos las distintas **clases DAO**), además de clases orientadas a la **capa de datos** con métodos como **crearObjeto()**, **encontrarObjeto()**, **borrarObjeto()**, etc.

- **Fase 2 (dentro de la app):** Los objetos se transforman para la base de datos...

  Llamada también de **persistencia o mapeo**. Contiene los siguientes **agentes**:

  • **Proveedor JPA**: librería que hace posible toda la funcionalidad de JPA: **javax.persistence**.

  • **Archivo de asignación**: es un **fichero XML** donde se almacena la **configuración** de la asignación de los **datos de una clase JAVA** (**POJO**) y los **datos reales** de la base de datos relacional.

  • **JPA Cargador**: realmente esta parte funciona como una **memoria caché**, **cargará los datos** de la base de datos proporcionando algo parecido a una **copia**; para de esa forma, realizar **interacciones rápidas** con las clases de servicio.

  • **Reja de objeto**: es el lugar donde se **almacenan temporalmente una copia** de los datos de nuestra base de datos relacional. Se le llama objeto **grid**, por lo que todas las **consultas pasarán por este punto**, y una vez realizadas las **verificaciones** pasará a la **base de datos** principal.

- **Fase 3 (exterior de la app – BBDD extrerna)**:

  Llamada **fase de datos relacionales**. Una vez pasada la reja de objetos y todo haya ido bien, se irá directamente **a base de datos**. **Hasta entonces** como hemos mencionado antes, se permanecerá en ese **espacio temporal de caché**.

Esquema arquitectura ORM

FASE 1	FASE 2	FASE 3
Capa servicio	Object Grid	BDD
POJO	JPA Loader	
Entidades	Archivo asign.	
DAO	JPA provider	
	Mapping	

Herramientas ORM

Los **ORM** más **usados en Java**:

• **EBEAN**:

Algunas características son:

• Soporte en bases de datos: soporta bases de datos como **H2, Postgres, Mysql, NuoDB, PostGis, MariaDB, SQL Server, Oracle, SAP**, etc.

• Múltiples niveles de abstracción: **consultas** de tipo **ORM** mezcladas con **SQL**, además de consultas **DTO** (Data Transfer Object).

• **Beneficios**: evita automáticamente N+1, usa **caché** de tipo **L2** para reducir carga de base de datos, realiza consultas **mezclando base de datos y cacheado L2**, ajusta automáticamente las consultas ORM, contiene **tecnología Elasticsearch para caché L3**, etc. Elasticsearch es un motor de búsqueda y analítica de RESTful distribuido capaz de abordar un número creciente de casos de uso.

• **IBATIS**:

Un ORM que aparece de la mano de **Apache Software Foundation**. En 2010 el desarrollo se centralizó en "Google Code", usando el nuevo nombre MyBatis. Posee **soporte para Java y .Net**. Algunas de sus características son:

• Posee la opción de **dividir** la **capa de persistencia** en:

capa de **abstracción**,
capa de **framework persistente**,
capa de **Driver**.

• Una de sus virtudes es la facilidad de **interactuar con los objetos** y los **datos** de las bases de datos **relacionales**.

• Ofrece **abstracción** a nivel de la **capa de persistencia** de **objetos**.

• **HIBERNATE**:

El ORM **más extendido y más usado**. Disponible para lenguaje Java y también **para .Net** (denominándose para éste, **Nhibernate**). Facilita el **mapeo relacional** de los distintos objetos entre una base de datos relacional y el modelo de objetos de la aplicación; para lo que se apoya en un **fichero .xml** que representa y establece dichas relaciones o también por medio de **anotaciones** donde se establecen las relaciones. Algunas de sus características:

• **Simplicidad**: al disponer de un solo fichero .xml para establecer las relaciones, es muy sencillo e intuitivo tanto dirigirnos a éste como consultar cualquier tipo de relación entre entidades o atributos.

• **Robusto**: dispone de muchas **características adaptadas al lenguaje Java**: colecciones, herencia, abstracción, orientación a objetos, etc. En la capa de abstracción ofrece una propia capa de **consultas SQL llamada HQL**, orientada a **facilitar la sintaxis** y a mejorar la **eficiencia** de estas.

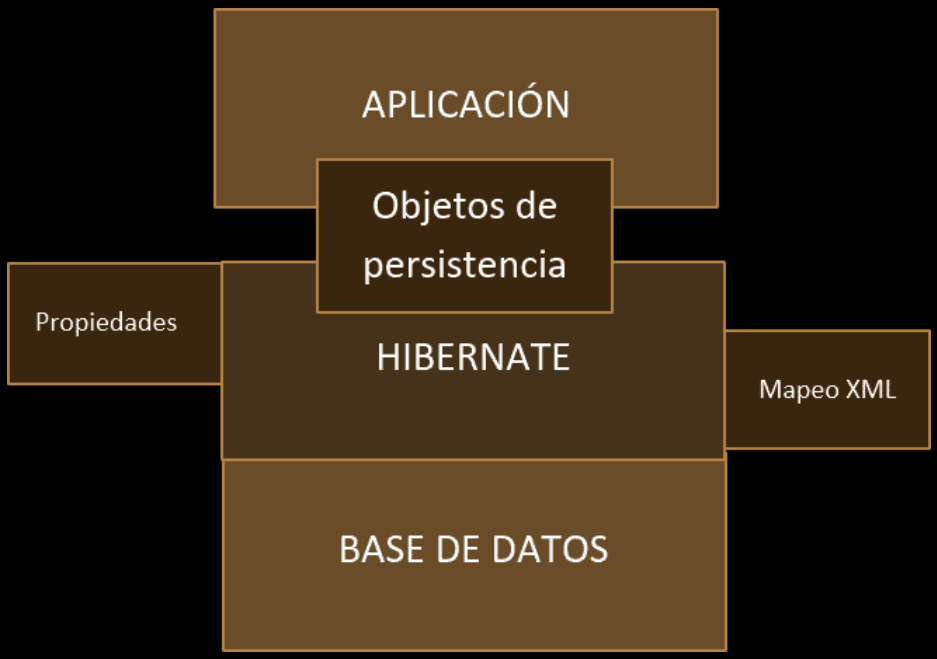
Definición de la arquitectura de Hibernate

Servicio de **alta eficiencia** de **mapeo** objeto-relacional y con **funcionalidades** bastante **potentes de consulta**. Empezaremos aprendiendo sobre su **arquitectura** y avanzaremos en la **instalación y configuración** de este Framework en aplicación Java.

La arquitectura de Hibernate incluye **distintos objetos** como son los objetos de **persistencia**, **sesiones**, **transacciones**, entre otros.

Usa el **principio** de **reflexión Java:** Permite el **análisis y la modificación** de los distintos **atributos** y características de las distintas clases **en tiempo de ejecución**.

Arquitectura Hibernate y diferentes capas que lo conforman:



Capas de **Aplicación e Hibernate unidas** por los **Objetos de persistencia:**

> porque en una **parte específica de la aplicación** se da cierta **conversión** (fichero de **mapeo**), dónde la información fluye y es mapeada desde dichos ficheros persistentes **a la base de datos**.

Capa **Hibernate:**

> **→** Realiza la **conexión con el driver**,
>
> → Se cargan:
>
> > **configuraciones Hibernate**,
> >
> > todas las **entidades** previamente diseñadas.

Componentes de Hibernate

Piezas clave que hacen **posible el mapeo objeto relacional**:

Algunos de sus **componentes principales** son:

- **SessionFactory Object**: mediante el que se permitirá el uso de **objetos** de tipo **Session**. Este objecto Java se puede instanciar de diferentes formas: normalmente **coge la configuración** existente en el **fichero de configuración** establecido por defecto. Utilizaremos un **objeto SessionFactory** por **cada base de datos** que tengamos en la aplicación.

- **Session Object**: utilizaremos dicho objeto para instanciar una **conexión directa** con nuestra base de datos relacional. Es un objeto no muy pesado y su función principal es **interactuar con la base de datos**. Este tipo de objetos **no deben permanecer abiertos mucho tiempo** por temas de **seguridad**.

- **Transaction Object**: es un objeto opcional, que básicamente **maneja las transacciones** directamente con las **bases de datos** relacionales. Si no se quiere hacer uso de dicho objeto, también **podemos indicar manualmente** aquellos **bloques** que queremos que sean **transaccionales**. Recordemos de temas anteriores la definición de transacción orientada al bloque de operación/es, cuyo objetivo consistía en **persistir** todas y cada una de las operaciones que contenían dicho bloque, **o** por el contrario, realizar **rollback** (marcha atrás) en dicha operación.

- **Query Object**: en Hibernate disponemos de varias formas de realizar **consultas** a la base de datos. Este tipo de objetos utilizan consultas de tipo SQL o de tipo **Hibernate Query** (**HQL**). Con este tipo de objetos **enlazaremos los distintos parámetros de nuestra consulta**, podremos realizar ciertas restricciones como controlar el número de resultados, y ejecutar la consulta. Es un modo de realizar consultas **mucho más dinámico que** las consultas **nativas**.

- **Criteria Object**: desaparecerá el lenguaje nativo de SQL para dar paso a las **consultas** por medio de **objetos Java**, y por medio de las funciones que nos ofrece Hibernate, que más tarde serán **traducidas a sentencias SQL**.

Instalación de Hibernate

Vamos con la instalación del framework **ORM** (Object Relational Mapping) que nos acompañará en el tema, y que nos ayudará a desarrollar ese concepto de **mapeo objeto – relacional**, denominado **Hibernate**, con **SpringBoot** en una **aplicación web Java:**

Dependiendo del **tipo de proyecto** Java que tengamos construido, podremos instalar Hibernate **de diferentes formas**.

Básicamente podremos añadir el .**jar a nuestro proyecto** con la versión específica de Hibernate que nos hayamos descargado. Si por el contrario estamos trabajando en un **proyecto web** con gestor de **dependencias maven**, podremos **agregar nuestra dependencia** de la **versión** determinada de **Hibernate**, y al **compilar**, se **descargará** dicha **librería** y tendremos las **clases del framework** disponibles para su uso.

Ahora, realizaremos la instalación agregando la dependencia determinada usando **Spring Boot**, considerando que estamos realizando una **aplicación desde cero**, a la cual queremos agregar Hibernate. Si éste **no fuese el caso**, nos podríamos quedar simplemente con la parte de **agregar la dependencia**.

Como ya hemos hecho anteriormente con las bases de datos embebidas, nos dirigiremos a la **web inicializadora** de **arquetipos** de **Spring Boot**. Hoy en día disponemos de la siguiente URL:

https://start.spring.io/

Una vez hemos ingresado en la URL, nos centraremos solo en la parte de añadir el framework que vamos buscando, ya que la composición del proyecto SpringBoot la vimos en temas anteriores. Hacemos clic en **Add dependencies** (añadir dependencias).

Buscando por Hibernate, añadiremos directamente **JPA con Sping Data**, e **Hibernate** y nuestras dependencias serían:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Aquí podemos observar la **dependencia agregada**. Si por el contrario quisiéramos agregar Hibernate **sin tener a Spring Boot** de mediador, podríamos dirigirnos a la web oficial de **Hibernate** y encontrar allí la **última versión para la dependencia** adecuada.

Configuración de Hibernate

Continuaremos con nuestra aplicación en Spring Boot y, por lo tanto, para realizar cambios de **configuración** nos dirigiremos al fichero "**application.properties**" de nuestro proyecto. A continuación, comentaremos algunos de los aspectos más importantes y usados en la configuración del framework instalado Hibernate.

Comunicación con base de datos:

```
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Con esta línea estamos estableciendo un nexo de unión entre nuestro framework Hibernate y la base de datos.

Una vez situados en el **fichero application.properties** dispondremos de la línea que hemos colocado en la arriba, comunicando nuestro **framework con la base de datos** que tenemos instalada. En este caso es una **base de datos H2**, pero si tuviéramos una base de datos distinta, cambiaríamos y **adaptaríamos la línea en función a la base de datos que tengamos**.

Con la siguiente configuración, estaremos indicando que **habilitamos las trazas de tipo SQL**, y además, que se muestren con el **formato SQL** correspondiente.

Trazas SQL:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Con las siguientes líneas de configuración, estaremos habilitando los **logs de Hibernate a true**; de esta forma podremos ver un **log** extenso sobre los **errores que se vayan dando**, así como las distintas **líneas de log** que **queramos** mostrar por nuestra cuenta. En la imagen también podemos observar que el **nivel de log** está establecido a **debug** (también podremos establecerlo a **nivel info**).

Logs:

```
spring.jpa.properties.hibernate.generate_statistics=true
logging.level.org.hibernate.type=trace
logging.level.org.hibernate.stat=debug
```

Con esta última línea, estaremos permitiendo a Hibernate **crear manualmente distintas entidades** que queramos generar en un fichero que habría que crear ("**schema.sql**"), y más tarde popular dichas tablas con otro script ("**data.sql**").

Inicialización BDD:

```
spring.jpa.hibernate.ddl-auto=none
```