Acceso a datos

ILERNA

1. Gestión de ficheros

En los siguientes temas aprenderemos todos los conceptos básicos que hay que tener en cuenta cuando necesitamos tratar con ficheros con lenguaje Java.

Para entender mejor qué trataremos en los siguientes apartados, primero deberemos entender bien los conceptos básicos: ¿qué podemos entender por un fichero?

> Un fichero es un archivo que contendrá un conjunto de caracteres o bytes que se almacenarán en el dispositivo en una ruta y con un nombre concretos.

Es el archivo que usará nuestro programa para almacenar, leer, escribir o gestionar información sobre el proceso que se esté ejecutando. Existen diferentes tipos de ficheros, como, por ejemplo:

- Fichero estándar: es un archivo que contiene todo tipo de datos: caracteres, imagen, audio, vídeo, etcétera. Normalmente son ficheros que contienen información de cualquier tipo.
- Directorios o carpetas: son ficheros que albergan más archivos en su interior. Su principal utilidad es mantener un orden o jerarquía en nuestros sistemas.
- Ficheros especiales: son todos esos ficheros que usa nuestro sistema operativo y que se utilizan para controlar los dispositivos o periféricos de nuestro ordenador.

En este tema profundizaremos en el tipo de ficheros estándar y en los directorios. Como explicaremos más adelante, este tipo de ficheros nos permitirán realizar diferentes acciones para tratar los ficheros y para mantener un orden y jerarquía con las carpetas.

Podemos destacar dos tipos de ficheros de datos:

- Los ficheros de bytes: también conocidos como ficheros binarios, son archivos que usan los programas para leer o escribir información.
- Los ficheros de caracteres: también conocidos como ficheros de texto, nos permitirán leer o escribir la información que contengan.

Un fichero se caracteriza por estar formado por la ruta en la que está almacenado, el nombre y una extensión, siguiendo este orden. Además, tenemos que tener en cuenta

que no podrán existir ficheros con el mismo nombre, ruta y extensión. Para que sean únicos, el nombre o la extensión en la misma ruta deben ser distintos.

Para tener acceso a un fichero determinado, se utiliza una ruta (o también la podemos nombrar *path*) que indica la ubicación de ese fichero en nuestro sistema. La ruta está compuesta por diferentes niveles jerárquicos (carpetas) separado por un símbolo barra **/**, Aunque en Windows, para separar los niveles jerárquicos, se utiliza la contrabarra o ****. En cambio, en Unix el separador será **/**. Eclipse admite tanto **/** como **** cuando definimos la ruta.

Si queremos definir la ruta independientemente del sistema operativo, podemos realizarlo de este modo:

```
//Ejemplo con la ruta directa al string
File archivoNoseguro = new File("carpeta/ejemplo.txt");
//Ruta que asegura el separador correcto segun plataforma
File archivo = new File("carpeta"+File.separator+"ejemplo.txt");
```

Existen dos tipos importantes de rutas que nos serán muy útiles en la gestión de ficheros:

- Ruta absoluta: se conoce como la ruta desde la carpeta padre:

  C:/Ilerna/accesoDatos/tema1/ejercicio.txt

- Ruta relativa: es aquella que coge como referencia el directorio actual para dar la ruta. La diferencia entre la ruta absoluta y la relativa es que no se indica la carpeta padre u origen y solo se da la guía desde la carpeta actual. Se indica con un punto, una barra y el nombre de los diferentes directorios separados por barras. Teniendo en cuenta que la carpeta actual sea *accesoDatos*, veamos este ejemplo:

  . /tema1/ejercicio.txt

La extensión del archivo nos permitirá diferenciar qué programa puede utilizar ese fichero. Se considera extensión todo lo que podemos encontrar después del punto que ponemos al final de nombre. Veamos el ejemplo:

ejercicio.txt → la extensión será el .txt

ejercicio.doc → la extensión será el .doc

El fichero se guardará según la codificación del dispositivo que estemos usando.

Los archivos que trataremos, en muchas ocasiones, contendrán información de texto o caracteres. Cada lengua utiliza un tipo de carácter distinto de otra, por ejemplo, el ruso utiliza un abecedario diferente que el español, por lo que usará caracteres distintos. Los caracteres se almacenan en nuestro ordenador como uno o más bytes.

Básicamente, podemos asumir que todos los caracteres están almacenados en ordenadores usando un código especial, es decir, una codificación de caracteres proporciona una clave para descifrar el código. Es un conjunto de asignaciones entre los bytes de los ordenadores y los caracteres en el conjunto de caracteres. Sin la clave, cuando el ordenador descifre los caracteres de ese fichero, aparecerán sin descifrar y se verán raros. Por ejemplo, algo así:

H‰ÄTMoÓ@¼ï¯xGûàõ~z×¨ªÔ´——————————————————————
©¨(–8¤=Ç)AÔn£Â¿ç½õ¦IH‘ƒ³Ÿ3ãyã—

Este sería un ejemplo de mala interpretación de un *encoding*. Los caracteres no son legibles y no podemos interpretar la información.

Se denomina ***encoding*** al sistema utilizado para transformar los caracteres que usa cada lenguaje en un símbolo que un ordenador pueda interpretar.

La codificación de caracteres asigna los caracteres escogidos a bytes específicos en la memoria del ordenador, y luego, para mostrar el texto, lee los bytes nuevamente en caracteres. Principalmente, se basa en crear tablas de equivalencias entre caracteres de lenguaje entendible por las personas con su correspondencia al lenguaje que usa un sistema informático.

No es necesario saberse todos los que existen, pero aquí os mostraremos los más importantes:

ASCII

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Ilustración 1. Tabla ASCII.

Es el conjunto de caracteres creado por la American National Standard Code for Information (ANSI), en 1967. Codifica caracteres, letras y símbolos que usamos día a día.

ISO-8859

Se trata de otro tipo de *encoding* bastante conocido que se caracteriza por incluir letras, símbolos y caracteres, pero además también los acentos y los símbolos de interrogación y exclamación. Este tipo de codificación utiliza 8 bits, por tanto, tiene una capacidad de 256 caracteres, lo que la hace más amplia que ASCII. Incluye los 128 caracteres de ASCII, pero se añaden símbolos matemáticos y letras griegas, entre otros. Con el tiempo, este *encoding* se ha quedado corto en cuanto a contemplar los diferentes alfabetos de distintos idiomas. Por este motivo, se han ido creando diferentes especializaciones de esta codificación. Podemos encontrar también estos otros:

- ISO 8859-1 (Latin-1), para la zona de Europa occidental.
- ISO 8859-2 (Latin-2), para la zona de Europa occidental y Centroeuropa.
- ISO 8859-3 (Latin-3), para la zona de Europa occidental y Europa del sur.
- ISO 8859-4 (Latin-4), para la zona de Europa occidental y países bálticos (lituano, estonio y lapón).
- ISO 8859-5, para el alfabeto cirílico.

- ISO 8859-6, para el alfabeto árabe.
- ISO 8859-7, para el alfabeto griego.
- ISO 8859-8, para el alfabeto hebreo.
- ISO 8859-9 (Latin-5), para la zona de Europa occidental con los caracteres del alfabeto turco.
- ISO 8859-10 (Latin-6), para la zona de Europa occidental, incluye los caracteres del alfabeto nórdico, lapón y esquimal.
- ISO 8859-11, incorpora caracteres del alfabeto tailandés.
- ISO 8859-13 (Latin-7), incorpora caracteres para los idiomas bálticos y el polaco.
- ISO 8859-14 (Latin-8), incorpora caracteres para los idiomas celtas.
- ISO 8859-15 (Latin-9), añade el símbolo del euro.
- ISO 8859-16, incorpora caracteres para los idiomas polaco, checo, eslovaco, húngaro, albano, rumano, alemán e italiano.

Unicode

Es una norma de codificación creada en 1991 para unificar los tipos de codificación. Como hemos visto, existen multitud de variantes de codificación para abarcar diferentes idiomas. La creación de este *encoding* pretendía organizar en un mismo estándar los diferentes caracteres dentro de una misma codificación, para poder abarcar diferentes idiomas tanto de alfabetos europeos como de chinos, japoneses, coreanos o lenguas ya extinguidas con alfabetos diferentes. Para realizar las tablas de equivalencias, Unicode asigna un identificador numérico a cada carácter, pero también irá acompañado de información como la direccionalidad, la capitalización y otros atributos. Nuestro ordenador, según su arquitectura, utilizará diferentes bloques de 8, 16 o 32 bits para interpretar y representar los números. Estos tres diferentes bloques han creado diferentes codificaciones:

- UTF-8.
- UTF-16.
- UTF-32.

Hoy en día, la codificación más usada es la codificación de caracteres UTF-8.

Tanto los editores de texto, como los IDE (programa para desarrollar nuestra aplicación) normalmente dan la posibilidad de configurar qué tipo de codificación queremos usar.

Por ejemplo, en eclipse esta configuración se puede encontrar dirigiéndonos a *Window > Preferences > General > Workspace*. Se nos abrirá una ventana y abajo a la derecha podremos escoger la codificación, tal y como se muestra en esta captura:

Para que se haga efectivo este cambio, tendremos que darle a *Apply and Close*, y todos los ficheros que creemos a partir de este momento serán con este tipo de codificación.

En el caso de que queramos usar diferentes *encodings*, será necesario usar un fichero binario, es decir, un fichero que almacenará bytes con la información.

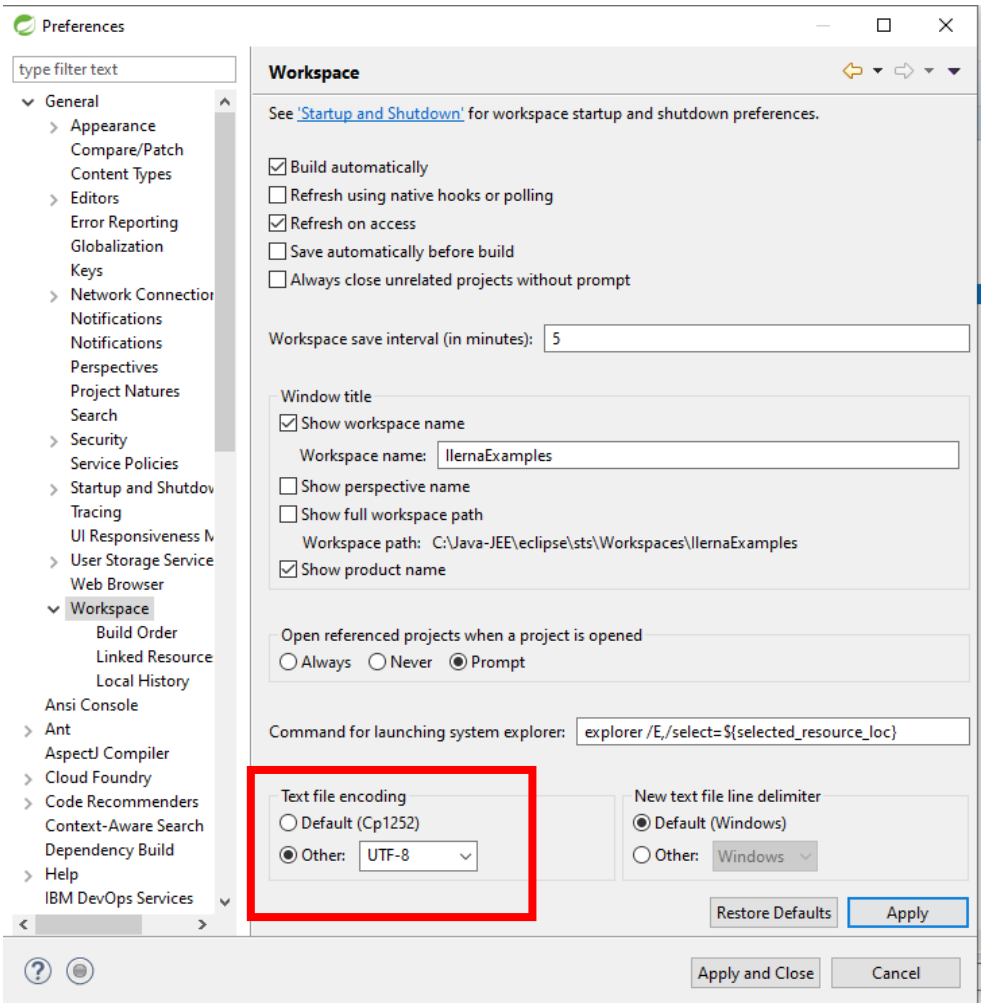


Ilustración 2. Captura de pantalla donde se muestra la configuración de Eclipse para modificar el encoding.

1.1. Clases asociadas a las operaciones de gestión de ficheros (secuenciales, aleatorios) y directorios: creación, borrado, copia, movimiento, entre otros

En este apartado, nos centraremos en la gestión de ficheros y directorios. Cuando se programa con Java, podemos realizar tareas básicas de gestión de ficheros que nos serán útiles para realizar todo tipo de acciones con ese fichero, desde crear hasta leer, borrar, copiar o mover de sitio carpetas o archivos.

Para tratar los diferentes archivos, Java tiene diferentes paquetes que nos pueden ayudar a realizar las operaciones básicas con cualquier tipo de fichero, tal y como veremos a continuación. Los diferentes objetos que veremos en esta sección pertenecen a diferentes paquetes de Java, algunos de la librería propia de Java, pero también hay otros que son necesario importación. Pero antes de complicarnos, mostraremos los más básicos e importantes.

El paquete más utilizado en el lenguaje Java es el paquete java.io. Dentro, podremos encontrar diferentes opciones que nos permitirán diferentes acciones con ficheros: creación, borrado, lectura, escritura, movimiento y copia, entre otras. Este paquete nos permitirá tanto la creación de ficheros como de carpetas.

Para seguir de manera más dinámica los ejemplos del libro, tenéis a vuestra disposición un repositorio en GitHub con todos los ejemplos más básicos que encontraréis en el libro.

https://gitlab.com/ilerna/common/java

1.1.1. Creación de directorios o ficheros

1.1.1.1. Creación de ficheros

Para la creación de ficheros, Java usa el paquete java.io. La creación de ficheros es una de las tareas más fáciles que nos encontraremos en esta lección. Para la creación de archivos, podemos utilizar diferentes librerías. Las más usadas son:

java.io.File

Se trata de la librería más básica para la creación de ficheros en Java. Nos será útil si solo queremos obtener información de un archivo o de una carpeta. Este paquete está enfocado a la lectura por *streams*.

> Un *stream* es una librería de java.io que se utiliza para gestionar flujos de datos, ya sea en ficheros, *strings* o dispositivos. Se encarga de tratar de manera ordenada una secuencia de datos con un origen y un destino.

Los datos que se leen desde este paquete no se guardan en ningún sitio, es decir, no se guardan en caché. Si fuera necesario recorrerlos otra vez, un *stream* no nos sería útil, sino que necesitaríamos cambiar a un búfer, pero ampliaremos información más adelante en el tema.

Para crear un nuevo objeto, debemos declarar una nueva instancia del objeto *File*, al cual le pasaremos el nombre de la ruta, el nombre del fichero que queramos crear y la extensión en un *string*.

En esta tabla tenemos los constructores más importantes que podremos utilizar de la librería *File*.

Constructor	Descripción
File(File padre, String hijo)	Crea una nueva instancia *File* a partir de una ruta abstracta padre y una ruta abstracta hija.
File(String ruta)	Crea una nueva instancia de *File* al convertir el nombre de la ruta dada en un nombre de ruta abstracta.

File (String padre, String hijo)	Crea una nueva instancia de *File* desde una ruta padre a una ruta hija.
File(URI uri)	Crea una nueva instancia de *File* a la cual se le pasará una URI. Una URI es una secuencia de caracteres utilizada para la identificación de un recurso en particular.

Tabla 1. Tabla con los constructores más importantes de la clase File.

Para la gestión de ficheros, deberemos tener una idea básica de los métodos que existen para cada librería. Para *File*, tenemos estos como los más destacados y que nos serán muy útiles en nuestros desarrollos:

Tipo	Método	Descripción
File	createTempFile(String prefijo, String sufijo)	Crea un archivo vacío en un directorio temporal, usando un prefijo y un sufijo que le pasaremos por parámetro para definir el nombre.
boolean	createNewFile()	Método que crea un fichero nuevo, vacío y con el nombre que le hayamos pasado al constructor al crear una instancia nueva de *File*.
boolean	canWrite()	Comprueba si nuestro programa puede escribir datos en el archivo.
boolean	canExecute()	Comprueba si se puede ejecutar el archivo.
boolean	canRead()	Comprueba si se puede leer el archivo.
boolean	isAbsolute()	Comprueba si la ruta del fichero es absoluta.
boolean	isDirectory()	Comprueba si el *File* que hemos creado es o no una carpeta.

boolean	isFile()	Comprueba si la nueva instancia creada de *File* es un archivo o no.
String	getName()	Devuelve el nombre del archivo o directorio creado.
Path	toPath()	Nos devuelve un objeto *Path* con la información de la ruta absoluta del fichero que hemos creado.
URI	toURI()	Construye un objeto URI que representa la ruta abstracta del fichero creado.
boolean	mkdir()	Crea una carpeta con el nombre que le hemos pasado al constructor al crear una nueva instancia *File*.
boolean	delete()	Borra el fichero siempre que esté en la ruta especificada.

Tabla 2. Tabla con los métodos más importantes de java.io.File.

A continuación, veremos cómo crear un archivo *File* pasándole como parámetro al constructor la ruta absoluta, el nombre y la extensión del fichero que queremos crear. El constructor, lo que realmente necesita es un *string*, que será la ruta compuesta. Lo podemos entender mejor en este ejemplo:

```
File archivo = new
File("src/main/resources/crearFicheros/fichero.txt");
if(archivo.createNewFile()) {
      System.out.println("Creado el fichero "+ archivo);
}else {
      System.out.println("No se ha creado el fichero");
}
```

La *c:* representa el nombre de nuestro disco duro, *archivo* será el nombre y la extensión será *txt*. Para definir dónde acaba el nombre y empieza el archivo, ponemos un punto. El método *createNewFile()* devuelve un booleano *true* si se crea el fichero, tal y como hemos visto en la tabla de métodos importantes.

Si queremos crear el fichero en una carpeta, deberemos ponerlo así:

```
File archivo = new
File("src/main/resources/crearFicheros/fichero2.txt");
if(archivo.createNewFile()) {
      System.out.println("Creado el fichero "+ archivo);
}else {
      System.out.println("No se ha creado el fichero");
}
```

Como vemos, después de la *c:* pondremos una / el nombre de la carpeta y otra /. Tendremos que tener en cuenta que la carpeta siempre debe existir antes de crear ese fichero. Si queremos crear una carpeta, veremos en el siguiente apartado cómo crearla.

java.nio.file.Files

Es una librería importante que encontraremos en la versión de Java 8. Es otra alternativa para la creación de ficheros, una de las más recomendadas. Tiene un gran repertorio de métodos para la creación, copia, borrado, escritura y lectura de datos.

Entrando en detalle de este paquete, nos permite un enfoque diferente a la hora de gestionar los datos: java.nio.file permite el almacenaje de la información del fichero en un búfer.

Un búfer es un bloque de memoria que permite almacenar temporalmente los datos y recorrerlos tantas veces como se desee para tratarlos.

Tal y como hemos comentado en la definición de búfer, los datos que se guardan en él se pueden volver a leer según se necesite. Nos proporciona un poco más de flexibilidad a la hora de tratar con un fichero.

Esta librería consta con métodos para la manipulación de ficheros que nos serán realmente útiles.

Tipo	Método	Descripción
Path	createDirectory(Path ruta)	Crea un directorio en la ruta indicada.
Path	createFile(Path ruta)	Crea un fichero en la ruta indicada.

Path	copy(Path origen, Path destino)	Crea una copia de un fichero origen a una ruta destino que le indiquemos.
Path	createTempDirectory(String prefijo)	Crea una carpeta temporal en la aplicación con el nombre que le pasemos como *string* al llamar al método.
Path	createTempFile(String prefijo, String sufijo)	Este método se encarga de crear un fichero temporal en la carpeta temporal del programa, usando el prefijo y el sufijo que le hemos indicado por parámetro.
void	delete(Path ruta)	Borra un fichero de la ruta indicada.
boolean	deleteIfExists(Path ruta)	Borra un fichero siempre que exista en la ruta indicada.
boolean	exists(Path ruta)	Comprueba si el fichero que le indicamos existe en la ruta indicada.
boolean	isDirectory(Path ruta)	Comprueba si ese fichero indicado en la ruta es una carpeta o no.
long	size(Path ruta)	Devuelve el tamaño del fichero que le indiquemos en la ruta.
Path	walkFileTree(Path ruta)	Este método se encarga de recorrer el árbol de directorios de una ruta recursivamente.

Tabla 3. Tabla con los métodos más útiles de la librería java.nio.File.

En este ejemplo, veremos mejor cómo crear un fichero con este paquete de Java.

```
Path ruta =
Paths.get("src/main/resources/crearFicheros/file.txt");
Path ejemploArchivo = Files.createFile(ruta);
System.out.println("Hemos creado un fichero "+ ejemploArchivo);
```

Esta vez, no utilizamos un *string* para definir la ruta del fichero que queremos crear, sino que utilizamos el objeto *Path*. Este es un objeto que se utiliza para localizar un fichero dentro de un sistema de archivos y se encargará de representar una ruta del fichero de nuestro sistema. La librería *Files* necesitará que se le pase este objeto para crear el fichero. Para crear el fichero, llamaremos al método *createFile()*, el cual devuelve un *path* con la ruta del fichero creado.

En el caso de que el fichero ya exista, Java lanzará un error. Al definir el *path*, tenemos que asegurarnos de que la ruta absoluta que utilizamos sea correcta, si no es así, se producirá un error cuando el programa ejecute esta parte del código porque no encontrará la ruta de carpetas indicada.

1.1.1.2. Creación de directorios

Para la creación de directorios se usa la misma clase *File* que hemos comentado anteriormente, pero el método para crearlo es algo distinto a la creación de ficheros. Si nos fijamos en el ejemplo:

```
File archivo = new File("src/main/resources/crearFicheros
directorio");
if(archivo.mkdir()) {
      System.out.println("Creado el directorio "+ archivo);
}
```

Como podemos ver, para crear un directorio el objeto *File* tiene un método que permite la creación de directorios: *mkdir()*. En el caso de que se realice con éxito, el método devuelve un booleano *true*.

Algunos de los problemas que nos podemos encontrar con este método son que, si no existe la carpeta indicada, o si la ruta absoluta que utilizamos para crear nuestro directorio no está bien, dará un error en la ejecución de nuestro programa. Debemos tener en cuenta esta posibilidad y controlar los errores.

```
String nombre = " src/main/resources/crearFicheros
carpetaEjemplo";
Path ruta = Paths.get(nombre);
Files.createDirectories(ruta);
```

El objeto **java.nio.file.Files** también nos ofrece posibilidad de creación de directorios a través del método *createDirectories()*, al cual le tendremos que pasar la ruta, como en

otros objetos. Aquí podemos apreciar un ejemplo de cómo usar este objeto con el método de creación de carpetas.

Este tipo de tareas son especialmente útiles para diferenciar carpetas y tener organización dentro de un ordenador o un SFTP para gestionar archivos que se van generando.

Un SFTP (protocolo de transferencia de archivos) es un programa estandarizado para transferir archivos entre ordenadores de cualquier sistema operativo.

Es un protocolo cliente servidor que será útil para transferir archivos que, por ejemplo, un usuario sube en una página web y que queremos transferir a un servidor para que queden almacenados. Una de las posibles utilidades sería crear carpetas por día, y crear nuevos archivos en esa carpeta.

1.1.2. Borrado

Para el borrado de archivos, tenemos métodos disponibles tanto en la librería java.io como en **java.nio.Files**. Estas dos librerías nos permitirán el borrado de ficheros y de directorios. A continuación, veremos los métodos más usados tanto en ficheros como en carpetas a modo de ejemplo.

En el objeto *File*, tenemos dos métodos que nos serán realmente útiles para el borrado:

- ***delete()****:* borrará el fichero que le indiquemos según la ruta o lanzará un error si no lo encuentra.

```
File archivo = new
File("src/main/resources/crearFicheros/fichero.txt ");
if(archivo.delete()) {
      System.out.println("Hemos borrado un fichero");
}
```

Como podemos ver, no es necesario crear un nuevo objeto para borrar el fichero en cuestión, solo necesitamos indicarle la ruta y el nombre del fichero, aunque también existe la posibilidad de borrar un fichero ya creado con el mismo método. En Java hay muchas maneras de poder borrar ficheros, os mostramos las más utilizadas.

- ***deleteOnExit()****:* este método, en cambio, borrará el fichero o el directorio que le indiquemos por ruta absoluta cuando la máquina virtual finalice.

```
File archivo = new File("src/main/resources/crearFicheros
fichero4.txt");
archivo.deleteOnExit()) {
      System.out.println("Hemos borrado un fichero");
}
```

La librería java.nio.Files también nos ofrece la posibilidad de borrar ficheros, aunque el procedimiento es algo distinto. También cuenta con un método *delete()* para borrar los ficheros. Para realizar un borrado, se debe realizar de este modo:

Para el borrado de directorios, se puede usar los mismos métodos que hemos indicado anteriormente. En el caso de directorios, antes de borrar se debe comprobar que esté vacío, si no, no va a borrarse y se lanzara una excepción *NoSuchFileException*. Hablaremos de las excepciones más adelante.

El objeto **java.nio.files.File** también ofrece posibilidad de borrado de carpetas.

```
String nombrePath = "
src/main/resources/crearFicheros/directorio";
Path carpeta = Paths.get(nombrePath);
Files.delete(carpeta);
```

```
//Primero crearemos una carpeta, en este caso se crea porque no
existe y sino nos dara error.
String nombrePath = "
src/main/resources/crearFicheros/directorio";
File ficheroCarpeta = new File(nombrePath);
ficheroCarpeta.mkdir();
//Indicamos la ruta con el directorio y el fichero que queremos
crear
String nombrefichero = " src/main/resources/crearFicheros
/ejercicio/fichero.txt";
File ejemplo = new File(nombrefichero);
ejemplo.createNewFile();

//Cogemos la ruta y lo borramos. Si no existe dara error
Path path = Paths.get(nombrefichero);
try {
      Files.delete(path);
}catch (IOException e) {
      System.out.println("No existe la carpeta" + ficheroCarpeta);
}
```

1.1.3. Copia

Otra de las acciones que podemos realizar con los ficheros o directorios es el copiado. Este método nos será útil para copiar ficheros de un origen a un destino, y tratarlos sin modificar los datos de origen.

Tal y como hemos mostrado en anteriores temas, veremos ejemplos de los dos objetos Java más utilizados que disponen del método para copiar ficheros.

Para empezar, la manera más fácil de copiar un archivo es con la API de java.nio. El objeto *Files* utiliza el método *copy()*, el cual necesita que se le pase por parámetro la ruta de origen y la ruta de destino, con un objeto *Path*. La copia fallará si el fichero destino ya existe, pero si utilizamos la opción *REPLACE_EXISTING* el fichero se sobrescribirá. Veremos en el ejemplo cómo usar esta opción.

Este método también se usa para los directorios, pero debemos tener en cuenta que solo se copiará la carpeta, no el contenido de esa carpeta.

```
Path origen =
Paths.get("src/main/resources/copiarFicheros/ejemploCopia.txt");
Path ejemploOrigen = Files.createFile(origen);
Path destino = Paths.get("src/main/resources/copiarFicheros/
/destino");
Path ejemploDestino = Files.createFile(destino);
Files.copy(origen, destino, StandardCopyOption.REPLACE_EXISTING);
```

Por otro lado, el objeto *Files* que se encuentra en la API java.io ofrece también la opción de copiar ficheros. Actúa de una manera parecida, pero se implementa de modo distinto. Para realizar el copiado, es necesario crear búferes.

Aquí podemos ver un ejemplo práctico:

```
//Primero se crean los ficheros
File archivoOrigen = new
File("src/main/resources/copiarrFicheros/origen.txt");
archivoOrigen.createNewFile();
File archivoDestino = new
File("src/main/resources/copiarFicheros/destino.txt");
archivoDestino.createNewFile();

try {
      // Se lee el origen
      InputStream origen = new BufferedInputStream(new
FileInputStream(archivoOrigen));
      // Fichero destino
      OutputStream destino = new BufferedOutputStream(new
FileOutputStream(archivoDestino));

      byte[] buffer = new byte[1024];
      int lengthRead;
      while ((lengthRead = origen.read(buffer)) > 0) {
            // se escriben los datos de un fichero a otro
            destino.write(buffer, 0, lengthRead);
            // Se cierra el proceso
            destino.flush();
      }

      // Ejemplo de copiado de datos con la api java.nio
      Path orig =
Paths.get("src/main/resources/copiarFicheros/ejemploCopia.txt");
      Path ejemploOrigen = Files.createFile(orig);

      Path dest =
Paths.get("src/main/resources/copiarFicheros/destino");
      Path ejemploDestino = Files.createFile(dest);

      Files.copy(orig, dest,
StandardCopyOption.REPLACE_EXISTING);
      System.out.println("se ha realizado la copia de ficheros");
} catch (Exception e) {
      e.getCause();
}
```

Tal y como vemos en el ejemplo, primero tendremos que crear un archivo origen y otro destino, con las rutas correspondientes. A continuación, necesitamos crear un *InputStream* con un nuevo *BufferedInputStream*, que se encargará de leer los datos del archivo de origen. Profundizaremos más adelante en estas clases, pero las clases *InputStream* y *BufferedInputStream* se caracterizan por leer archivos, y en este ejemplo nos serán útiles para coger el contenido del fichero origen y traspasarlo al fichero

destino. En cambio, las clases *OutputStream*, *BufferedOutputStream* y *FileOutputStream* se encargan de escribir ficheros.

Para el objeto destino, debemos crear un *OutputStream* que contendrá un *BufferedOutputStream* para albergar el resultado de la copia. Con un bucle, recorreremos el fichero origen dato a dato y lo escribiremos al objeto destino. Al finalizar, se llamará al método *flush()* para terminar con el proceso. Este método se encargará de vaciar el *OutputStream* y se guardarán los archivos de salida en un búfer.

Para realizar la copia de los archivos es necesario muchas más líneas de código en comparación con el otro ejemplo, por eso recomendamos usar la API java.nio.

1.1.4. Movimiento

Java proporciona funciones para mover ficheros entre carpetas. Hay diferentes maneras de hacerlo, y veremos las posibilidades que tenemos según la API que queramos usar.

La API java.nio es la que ofrece una opción más rápida para mover un fichero. Usa el método *move()*, al cual le tendremos que pasar por parámetro el origen e indicarle un destino. Como se hacía en la copia, tenemos la posibilidad de indicarle por parámetro *REPLACE*_EXISTING. Este parámetro le indica al método que, si existe un dichero con ese nombre, lo deberá sobrescribir. Aquí tenemos un ejemplo de cómo se debe implementar:

```
Path destino = Files.move(Paths.get("c:/ejercicio/ejemplo.txt"),
Paths.get("c:/ejercicio/destinoEjemplo.txt"),
StandardCopyOption.REPLACE_EXISTING);
```

Otra manera de realizar esta acción es con el objeto *File* de la API java.io. Este objeto tiene una forma más rudimentaria de realizar el movimiento. Básicamente, se encarga de renombrar el fichero a uno nuevo y borrar el fichero de origen. Recomendamos por su rapidez usar el objeto *Files*.

```
File origenArchivo = new
File("src/main/resources/crearFicheros/carpetaEjemplo/ficheroOrigen.txt");
origenArchivo.createNewFile();
if(origenArchivo.renameTo(new
File(src/main/resources/crearFicheros/destino.txt"))){
      origenArchivo.delete();
      System.out.println("Se ha movido el fichero.");
} else{
      System.out.println("Se ha producido un error.");
}
```

1.2. Formas de acceso a un fichero de texto en modo de acceso secuencial y aleatorio. Ventajas e inconvenientes de las distintas formas de acceso

En este apartado, explicaremos las diferentes maneras de acceder a la información de un fichero. Podemos diferenciar dos tipos de acceso: el acceso secuencial y el acceso aleatorio.

Para empezar, aprenderemos los conceptos básicos sobre el acceso a ficheros de manera secuencial.

Un archivo con acceso secuencial es un fichero donde se guarda la información en una secuencia de caracteres, de manera que el acceso a ellos se debe realizar en estricto orden, con base en el campo clave de su origen.

Para simplificar más la definición, son archivos donde se guardan los registros en orden, con base en el campo clave de origen. Para leer los datos de un fichero secuencial, se debe acceder a los datos uno después de otro y, una vez consultados, no se podrá acceder a ellos si no se sigue el orden. Aquí tenemos la simplificación a modo de esquema para aclarar el concepto.

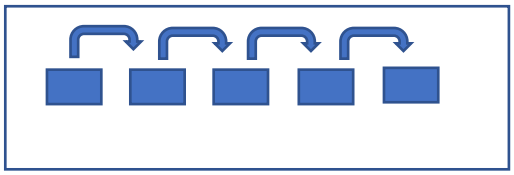


Ilustración 2. Acceso secuencial a ficheros.

Como se mencionó anteriormente, estos son archivos en los que los registros se almacenan en orden por el campo clave de registro. Primero demostramos archivos de acceso secuencial utilizando archivos de texto, lo que permite al lector crear y editar rápidamente ficheros fáciles de leer. Más adelante, veremos ejemplos prácticos de lectura y escritura de este tipo de acceso de datos.

Acceso archivo aleatorio

El acceso aleatorio a archivos es un tipo de acceso a datos que permite al programa Java acceder a los datos sin un orden, es decir, a cualquier posición en que se encuentren los datos, sin ningún orden.

A diferencia del acceso secuencial, con el acceso aleatorio no es necesario empezar desde la primera línea del fichero. Se asemeja a los *arrays* de bytes, ya que podemos acceder a cualquier parte de los datos con un puntero de fichero, es decir, indicando al método de acceso en qué posición queremos empezar a tratar los datos. En la siguiente ilustración, podremos apreciar más fácilmente a qué nos referimos con acceso aleatorio de los datos.

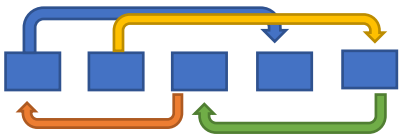


Ilustración 4. Acceso aleatorio a ficheros.

Como podemos apreciar, el acceso a los datos es totalmente aleatorio y sin ningún orden, y este tipo de acceso es realmente útil en aquellas aplicaciones de baja latencia que necesitan persistencia. Este tipo de acceso a datos es especialmente útil para situaciones en que el programa ha sufrido un error y es necesario acceder otra vez de manera aleatoria a los datos.

Java contiene un objeto que permite este tipo de acceso de datos: es el objeto *RandomAccesFile*. Este objeto tiene la habilidad de leer y escribir cualquier tipo de dato en un fichero de manera aleatoria. Cuando lee el contenido del archivo, comienza con la ubicación actual del puntero del archivo y el puntero avanza más allá de cuántos bytes se leen.

De manera similar, cuando escribe datos en un archivo de acceso aleatorio, comienza a escribir desde la ubicación actual del puntero del archivo y luego avanza el puntero del archivo más allá del número de archivos escritos.

Los métodos más destacados de este objeto son:

- *seek():* logra configurar el puntero del archivo a cualquier ubicación aleatoria.
- *getFilePointer():* permite obtener la ubicación donde se encuentra en ese momento.

Como veremos en los ejemplos más detenidamente:

```
//Paso 1: Crearemos un fichero donde se escribiran los datos
File archivoEjemplo = new File("src/main/resources/crearFicheros/
randomAccesFileEjemplo.txt");
try {
      if(archivoEjemplo.createNewFile()) {
            System.out.println("Se ha creado el fichero");
      }
} catch (IOException e1) {
      System.out.println("Se ha producido un error");
      e1.printStackTrace();
}
//Paso 2: Definiremos el texto que queremos añadir al fichero
String texto = "Prueba de informacion acceso a datos";

int posicion = 10;
String datos = null;
try {
RandomAccessFile lectura = new RandomAccessFile(archivoEjemplo,
"rw");
//Paso 3: Se mueve a la posición del fichero que le indiquemos
      lectura.seek(posicion);
//Paso 4: Aquí leemos el string de datos desde el objeto
RandomAccesFile
      datos = lectura.readUTF();
      lectura.close();
} catch (IOException e) {
      e.printStackTrace();
```

En el ejemplo, la clase *RandomAccesFile*, al crear una nueva instancia, necesita un objeto *File*. Para ello, primero crearemos un fichero en una ruta de nuestro sistema operativo. Después, también un *string* con el texto que queramos escribir en ese fichero, como también un *int* con la posición que queremos empezar a escribir el fichero. Seguidamente, declararemos una nueva clase *RandomAccesFile* y crearemos una nueva instancia pasándole el archivo *archivoEjemplo* y le diremos que queremos permisos de escritura y lectura: para ello, al constructor le pasaremos el *string* "rw". La *r* equivale a *reading* (lectura en inglés) y la *w* a *writting* (escritura).

A continuación, tenemos que realizar la llamada al método *seek()*, que es el encargado de moverse dentro del fichero, y le tenemos que indicar la posición a la que queremos empezar. Seguidamente, se hace la llamada a *readUTF()*, que se encargará de leer todos los datos del fichero y meterlos en un *string*. Cuando termina el proceso, hacemos la llamada al método *close()*, que cerrará todos los procesos y se asegurará de que todo queda cerrado sin consumir más recursos dentro del programa.

Ventajas e inconvenientes de las distintas formas de acceso

Las ventajas del acceso de datos secuencial frente al aleatorio es que este tipo de estructura se puede usar para persistir datos de manera más simple.

Por otro lado, las desventajas son evidentes: no es una manera de acceder a los datos de manera eficiente. Nos obliga a acceder a los datos en orden, aunque el resto de datos no nos interesen.

En cambio, el acceso de datos aleatorio proporciona mucho más control y eficiencia a la hora de tratar la información. Por un lado, podemos elegir el punto exacto de lectura de datos, permitiendo ser mucho más rápido y ahorrar consumo de recursos en la aplicación. Como desventaja podemos destacar que puede ser un método de acceso un poco más lento en comparación con el secuencial.

1.3. Clases para gestión de flujos de datos de un fichero binario desde / hacia archivos

En la gestión de ficheros, unas de las principales acciones son la lectura y escritura de datos en ficheros. Antes de adentrarnos en cómo realizarlo, tendremos que identificar qué tipo de datos estamos tratando. En primer lugar, podemos tratar ficheros basados en caracteres o basados en bytes (o ficheros binarios).

Un fichero binario es un tipo de archivo que contiene información en cualquier tipo de codificación (o *encoding*) representado por ceros y unos (conocidamente como binario) para ser tratado y almacenado por un programa o sistema.

Un fichero de caracteres es aquel que contiene información de texto sin caracteres raros, que se encuentra con la codificación por defecto del sistema que está tratando los datos.

1.3.1. Escritura y lectura de datos

Una de las acciones más importantes y útiles de los ficheros es la posibilidad de leer el contenido y escribir en él. Para ello, en este apartado aprenderemos como se realiza con las dos API más importantes y detallaremos los objetos y métodos más útiles para poder llevarlo a cabo.

1.3.1.1. Lectura

El lenguaje Java permite más de una manera de implementar una lectura de un fichero: la lectura de ficheros de texto y la lectura de ficheros binarios.

Ficheros de texto

Para leer un archivo de caracteres en el *encoding* que venga por defecto, hay diferentes clases que vienen por defecto en Java y que son las que vamos a explicar y utilizar. Todas estas clases están definidas bajo el paquete java.io.

- *FileReader:* es una clase muy útil para leer archivos de texto utilizando la codificación del sistema operativo. Los constructores de esta clase usan el búfer con tamaño predeterminado.

```
//Utilizamos para leer el fichero un archivo en la
carpeta resources
FileReader fichero = null;
try {
      fichero = new
      FileReader("src/main/resources/ejemplo.txt");
} catch (FileNotFoundException e) {
      System.out.println("Se ha producido un error");
      e.printStackTrace();
}

int i;
try {
      while ((i=fichero.read()) != -1)
            System.out.print((char) i);
} catch (IOException e) {
      System.out.println("Se ha producido un error");
      e.printStackTrace();
}
```

En este ejemplo, vemos que el procedimiento es algo más sencillo. Primero, debemos definir un fichero creando un fichero *FileReader* y pasándole al constructor la ruta del archivo que queramos leer. A continuación, creamos un bucle *while* que nos va a permitir leer los datos del fichero y mostrarlos por la consola. Si nos fijamos, en este caso también controlamos la posibilidad de no encontrar el fichero y también si se produce un error durante el proceso de lectura. Esta clase lee los datos carácter a carácter hasta encontrar un -1, que es

el número que se utiliza para indicar que no hay más caracteres. Cada lectura recogerá un carácter único, es por este motivo que utilizamos *print* y no *printIn*.

- *BufferedReader:* lee ficheros de texto desde un *stream* de entrada de caracteres. Esta clase realiza una lectura de los datos mediante un búfer, lo que lo convierte en un método muy eficiente de lectura. Tiene la peculiaridad que permite definir el tamaño de su búfer o usar el tamaño predeterminado. En general, cada petición de lectura que realiza un *BufferedReader* hace que se realice una solicitud de lectura correspondiente del flujo de bytes. Por ese motivo, es aconsejable envolver un *BufferedReader* alrededor de cualquier lector cuyos métodos *read()* puedan ser costosos, como *FileReaders* e *InputStreamReaders*.

```
//Utilizamos para leer el fichero un archivo en la carpeta
resources
File file = new File("src/main/resources/ejemplo.txt");
//Creamos el buffer
BufferedReader reader;
try {
      //Envolvemos el archivo dentro de un file reader
      reader = new BufferedReader(new FileReader(file));
      String datos;
      //Imprimimos los datos por consola
      while ((datos = reader.readLine()) != null)
            System.out.println(datos);
} catch (FileNotFoundException e) {
      System.out.println("Se ha producido un error");
      e.printStackTrace();
}
```

Primero, definiremos el archivo que queremos leer y su ruta, puede ser la ruta absoluta o la relativa. En este caso, hemos utilizado un fichero que se encuentra dentro del proyecto. A continuación, se define el *BufferedReader* y se declara una nueva instancia envolviendo el fichero *File* en un *FileReader* y pasándole ese último al constructor de *BufferedReader*. Seguidamente, solo nos hará falta crear un *string* que contendrá los datos del fichero y recorrer con un *while* el *BufferedReader* línea por línea e imprimirlo por consola. Como vemos, controlamos la posibilidad de error con un *FileNotFoundException*.

- *Scanner:* se trata de una clase que analiza los ficheros de caracteres y permite analizar la información del fichero y clasificarla según su tipo. Esta clase divide los datos que recibe en *tokens* (un *token* es el elemento más pequeño de un programa) utilizando un patrón delimitador que por defecto coincide con los espacios en blanco. Los *tokens* obtenidos se pueden convertir en valores de diferentes tipos utilizando los métodos *next()*.

```
//Utilizamos para leer el fichero un archivo en la carpeta
resources
File archivo = new File("src/main/resources/ejemplo.txt");
Scanner lector = null;
try {
      lector = new Scanner(archivo);
} catch (FileNotFoundException e) {
      System.out.println("Se ha producido un error");
      e.printStackTrace();
}

//Usamos \\Z como un delimitador
lector.useDelimiter("\\Z");
System.out.println(lector.next());
```

En este ejemplo, primero, es necesario definir un archivo *File* con la ruta del fichero que queremos leer. En segundo lugar, vemos que es necesario definir el *Scanner* y envolver en un *try-catch* la creación de una nueva instancia. Si no se produce ningún error, llamando al método *next()* podremos visualizar por la consola el contenido del fichero. Debemos tener en cuenta que se tiene que controlar si nuestro programa no encuentra el fichero indicado, para ello lo hemos definido en el *catch*.

Ficheros binarios

Para la lectura de ficheros binarios (también conocidos como ficheros de bytes), archivos con caracteres especiales o ficheros de imagen, es necesario usar otro tipo de clases más específicas para ese tipo de acción. Todas estas clases están definidas bajo el paquete java.io.

Vamos a explicar los más destacables:

- *InputStream:* se caracteriza por ser una superclase abstracta que se encarga de leer un *stream* de bytes. Solo es capaz de leer un byte a la vez, por lo que lo hace una opción bastante lenta. Al ser una superclase, no es útil por sí misma, es por ese motivo que se usan sus subclases.

> Se considera una superclase a aquella clase padre de la que derivan diferentes clases, también conocidas como subclases. Las subclases derivan de la clase padre y heredan todas sus propiedades y métodos.

> Una clase abstracta es un tipo de clase que permite la declaración de métodos, pero no su implementación. La implementación será realizada por las clases que implementen esa clase.

Si queréis refrescar conceptos, podéis consultar la documentación de Java, donde se explica mucho más detallado:

https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

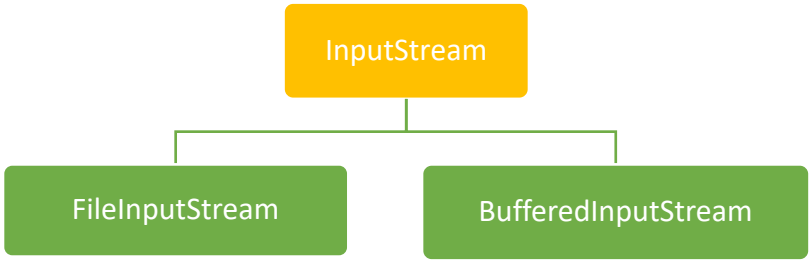


Ilustración 5. Esquema de la estructura de InputStream.

Esta clase tiene diferentes métodos que serán útiles si queremos leer datos de un *stream* de datos:

Método	Descripción
avaliable()	Este método se encarga de devolver los bytes disponibles en el *InputStream*.

close()	Es el método que se encarga de cerrar el *stream*.
mark()	Este método se encarga de marcar la posición de los bytes leídos del *stream*.
read()	Se encarga de leer los bytes de datos uno a uno.
read(byte[] array)	Se encarga de leer los bytes del *stream* y los almacena en un *array*.
reset()	Método que se encarga de quitar la marca de los bytes leídos por el método *mark()*.
skips()	Este método se encarga de descartar un numero especificado de bytes del stream que este leyendo en ese momento.

Tabla 4. Tabla de métodos importantes de InputStream.

- *FileInputStream:* es una subclase de *InputStream* que se utiliza para leer *streams* de bytes. Está enfocada al tratamiento de bytes sin procesar, como datos de imágenes, vídeo o audio. También puede leer ficheros de caracteres, pero no es el más adecuado, es mejor *FileReader*.

```
public static void lecturaFileInputStream() throws IOException{
	byte[] array = new byte[100];
	try {
	//Utilizamos para leer el fichero un archivo en la
carpeta resources
		InputStream archivo = new
		FileInputStream("src/main/resources/ejemplo.txt");

		//Lectura de bytes desde el Input Stream
		archivo.read(array);

		// Convierte los bytes en un string
		String datos = new String(array);
		System.out.println(datos);

		// Cerramos el inputStream
		archivo.close();
	} catch (FileNotFoundException e) {
		System.out.println("No se ha encontrado el
archivo");
		e.printStackTrace();
	}
}
```

En el ejemplo de arriba, primero hemos creado un *InputStream* con una nueva instancia de su subclase *FileInputStream* con la ruta relativa del fichero que se encuentra en nuestro proyecto, pero puede ser cualquier fichero de bytes. Para leer los datos del fichero, hemos implementado el método *read()*, que irá guardando los bytes en el *array* de bytes que hemos creado al principio. Para imprimir por consola los bytes, hemos creado un *string* pasándole el *array* de bytes. Una vez realizado todo esto, tendremos que cerrar el *stream* con el método *close()*.

- *BufferedInputStream:* es una clase que extiende (es decir, que implementa todos los métodos) de *InputStream* y que se utiliza para leer *streams* de datos en bytes de manera más eficiente. Esta clase se caracteriza por tener un búfer interno de 8192 bytes. Durante la operación de lectura, *BufferedInputStream* se encargará de leer una porción de bytes del fichero que se encuentra en el disco y almacenará los bytes en el búfer interno. Una vez almacenados en el búfer interno, se leerán los bytes individualmente.

```
try {

// creamos un InputStream para coger el fichero de la ruta de
nuestro proyecto
	FileInputStream fichero = new
FileInputStream("src/main/resources/ejemplo.txt");

// Creamos un BufferedInputStream y le pasamos el archivo al
constructor
	BufferedInputStream bufer = new
BufferedInputStream(fichero);

// Se encarga de leer el primer byte del fichero
	int i = bufer.read();
	while (i != -1) {
		System.out.print((char) i);
		// Va leyendo cada byte del buffer
		i = bufer.read();
	}
	bufer.close();
} catch (Exception e) {
	System.out.println("Se ha producido un error");
	e.getStackTrace();
}
```

En este ejemplo, podemos ver que hemos creado en primer lugar un fichero con una nueva instancia *FileInputStream* a la cual le pasaremos la ruta del fichero que queremos leer. A continuación, creamos un búfer *BufferedInputStream*, al cual

le pasaremos el fichero creado en la línea anterior a través del constructor. Seguidamente, tendremos que procesar el fichero, para ello utilizaremos el método *read()* del búfer e iremos leyendo byte a byte el contenido. Mientras el byte no equivalga a -1 se irá recorriendo el fichero e imprimiendo por consola el contenido. Una vez terminado el proceso, cerraremos el búfer. Todo el ejercicio está envuelto en un *try-catch* para controlar los posibles errores que puedan ocurrir, como que no encuentre el fichero o se produzca un error leyendo el búfer de bytes. Tenemos que tener en cuenta que siempre se deben controlar los errores

1.3.1.2. Escritura de datos

Para la escritura de datos, también podemos diferenciar dos grupos: el de escritura de ficheros de caracteres y el de escritura de ficheros de bytes. Primero detallaremos las clases de escritura de ficheros de caracteres. El proceso es muy parecido al de lectura de datos, pero se usarán las clases que detallaremos a continuación:

Ficheros de texto

Para la escritura de ficheros de texto, usaremos las clases que podemos encontrar en el paquete java.io.

- *Writer:* es una superclase abstracta que no se usa sola por sí misma, sino que siempre va acompañada de sus subclases, como hemos visto con *InputStream*.

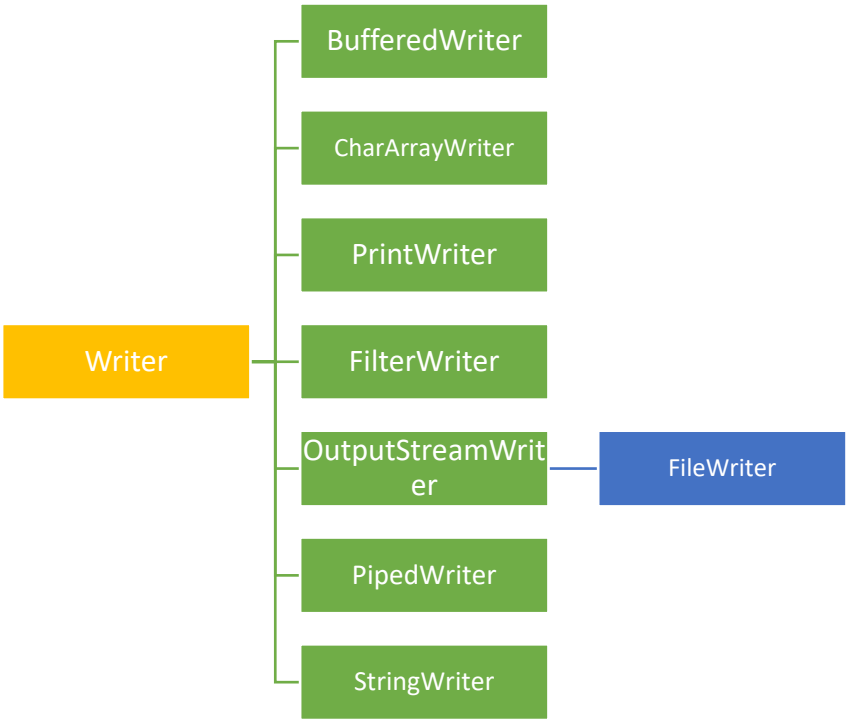


Ilustración 6. Esquema de las subclases de la clase Writer.

Estos son los métodos de la clase más destacables:

Método	Descripción
append(char c)	Este método añade el carácter que le indiquemos
close()	El método cierra el *stream* que está en uso.
flush()	El método obliga escribir toda la información que tenga el objeto *OutputStreamWriter* al destino correspondiente.
getEncoding()	Este método nos indica qué *encoding* se está utilizando para escribir los datos en el fichero.
write(char c)	Este método escribe en un fichero un carácter que le pasemos por parámetro.
write(char[] array)	Este método escribe en un fichero el *array* que le pasemos por parámetro.

write(String data)	Este método escribe en un fichero el *string* que le pasemos por parámetro.

Tabla 5. Métodos más destacables de Writer.

- *OutputStreamWriter:* es una subclase de *Writer* que se usa para convertir *streams* de caracteres, pero también se utiliza para *streams* de bytes. Como es capaz de tratar tanto caracteres como bytes, se usa como puente entre estos dos tipos de datos.

  A continuación, veremos un ejemplo de escritura de datos. Primero, es necesario la creación de un nuevo fichero *FileOutputStream*, creando una nueva instancia y pasándole al constructor la ruta del fichero que queremos escribir. Este objeto será un auxiliar que pasarle al constructor para poder crear un *OutputStreamWriter*. Seguidamente, para escribir información, deberemos tener un *string* con los caracteres que escribir y hacer una llamada al método *write()* pasándole por parámetro el *string*. Una vez finalizado el proceso, deberemos cerrar el *stream* con el método *close()*.

```
String data = "Ejemplo de escritura de datos con
FileOutputStream";

try {
      //Creamos un FileOutputStream
      FileOutputStream archivo = new
      FileOutputStream("src/main/resources/outputStream.
      txt");
// Creamos el stream que nos va a ayudar a escribir los
datos en el fichero indicado
      OutputStreamWriter escribirDatos = new
      OutputStreamWriter(archivo);
// Con este método escribiremos datos en el fichero
      escribirDatos.write(data);
// Cerramos la el writer
      escribirDatos.close();
}catch (Exception e) {
      System.out.println("Se ha producido un error");
      e.getStackTrace();
```

En este ejemplo, tendremos que realizar la operación con un *try-catch* para poder controlar los posibles errores que puedan producirse. Los más usuales siempre son un *FileNotFoundException* cuando no encuentra la ruta del fichero

indicado o un error en el proceso de escritura de los datos. Este método lanza un *IOException*, por lo tanto, tendremos que tenerlo controlado en un *catch*.

- *FileWriter:* es una clase que nos permitirá escribir caracteres en un archivo. Es una subclase de *OutputStreamWriter*.

```
String data = "Ejemplo de escritura de datos con
FileWriter";

try {
      // Creamos un FileWriter
      FileWriter output = new
      FileWriter("src/main/resources/fileWriter.txt");
      // Con este método escribiremos datos en el fichero
      output.write(data);
      // Cerramos la el writer
      output.close();
}catch (Exception e) {
      System.out.println("Se ha producido un error");
      e.getStackTrace();
}
```

El funcionamiento para escribir datos con esta clase es muy parecido al anterior, usan los mismos métodos, ya que las dos clases heredan métodos de la clase padre *Writer*. En primer lugar, deberemos definir la información que queremos escribir en el fichero. Crearemos un *FileWriter* definiendo la ruta del fichero que escribir. Seguidamente, haciendo uso del método *write()*, pasaremos por parámetro el *string* con la frase que hemos definido. Una vez escrito, cerraremos el *Writer* con el método *close()*.

- *BufferedWriter:* es una subclase de *Writer* que también permite almacenar datos en el búfer. Esta es la clase más eficiente para escribir datos en un archivo, ya que permite escribir los datos en el búfer y no en el disco. Una vez que el búfer esté lleno o se cierre, los datos se escriben en el disco.

  Como vamos a ver en el siguiente ejemplo, en primer lugar, deberemos crear un *FileWriter* creando una nueva instancia y pasándole al constructor la ruta del fichero que queremos usar. A continuación, crearemos el *BufferedWriter* y le pasaremos el archivo creado en la línea anterior. Seguidamente, podremos escribir los datos en el fichero y al acabar cerraremos el búfer. Como se puede apreciar, el proceso es igual al ser subclases de *Writer*.

```
String data = "Ejemplo de escritura de datos con
BufferedWriter";
try {
      // Creamos un FileWriter
      FileWriter file = new
      FileWriter("src/main/resources/output.txt");
      // Creates a BufferedWriter
      BufferedWriter output = new BufferedWriter(file);
      // Con este método escribiremos datos en el
fichero
      output.write(data);
      // Cerramos la el writer
      output.close();
}catch (Exception e) {
      System.out.println("Se ha producido un error");
      e.getStackTrace();
}
```

Ficheros binarios

La escritura de ficheros binarios consiste en escribir datos de tipo byte en un fichero. Este tipo de fichero podrá ser un fichero de datos codificado, un archivo de audio, uno de vídeo, una foto, etcétera.

Para la escritura de ficheros binarios, usaremos las clases que podemos encontrar en el paquete java.io.

- *OutputStream:* pertenece al paquete java.io y es una superclase abstracta que se utiliza para escribir *streams* de bytes.

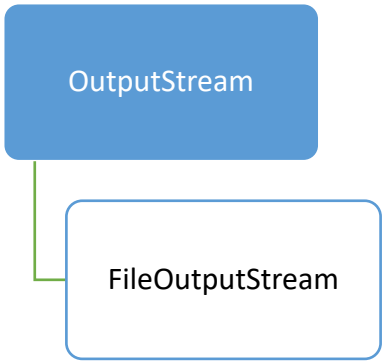


Ilustración 7. Esquema de las subclases de la clase OutputStream.

Estos son los métodos de la clase más destacables:

Método	Descripción
close()	El método cierra el *stream* que está en uso.
flush()	Este método libera los datos del *stream*.
write(int b)	Este método se encarga de escribir un byte indicado al fichero.
write(byte[] array)	Este método escribe todo el *array* en un fichero.

Tabla 6. Métodos más importantes de la clase OutputStream.

- *FileOutputStream:* esta clase se encargará de escribir *streams* de bytes en los ficheros. Es una subclase que hereda de *OutputStream*.

```
String data = "Ejemplo de escritura de datos con
FileOutputStream";
try {
	FileOutputStream output = new
	FileOutputStream("src/main/resources/escrituraBytes/file
	Output.txt");

	byte[] array = data.getBytes();

	//Escribimos los datos en el archivo
	output.write(array);

	// Cerramos el writer
			output.close();
} catch (Exception e) {
	System.out.println("Se ha producido un error");
	e.getStackTrace();
}
```

El procedimiento para usar las clases y sus métodos sigue la dinámica explicada ya en diferentes apartados. En primer lugar, crearemos una nueva instancia de la clase *FileOutputStream* y le pasaremos al constructor la ruta del fichero que queremos escribir. Como el método *write()* necesita un *array* de bytes para escribir los datos, transformaremos el *string* que queremos escribir en un *array* de bytes con el método *getBytes()* que tiene la clase *String*. A continuación, si no se produce ningún error, cerraremos el *stream* con el método *close()*.

- *ByteArrayOutputStream:* es una clase que se utiliza para envolver *OutputStream* para dar soporte a las capacidades del búfer. Se trata de una de las clases más eficientes de escritura de datos.

Estos son solo algunas de las más importantes, pero en realidad existen muchas más clases para escribir datos en archivos de texto. Aquí solo mostramos una muestra de lo que podemos llegar a hacer con Java, pero existen infinidad de posibilidades.

```
String data = "Ejemplo de escritura de datos con
BufferedOutputStream";

try {
	BufferedOutputStream bufer = new
	BufferedOutputStream(new
	FileOutputStream("src/main/resources/escrituraBytes/outp
	ut.txt"));

	bufer.write(data.getBytes());
	bufer.close();
} catch (IOException e) {
	System.out.println("Se ha producido un error");
	e.getStackTrace();
}
```

1.4. Trabajo con archivos XML

En este tema, trabajaremos con archivos XML. Para refrescar la memoria, los XML (lenguaje de marcas extensible) es un fichero de texto simple de metalenguaje extensible que consta de diferentes etiquetas que contienen los datos.

Podemos entender el concepto **metalenguaje** como el código que utilizaremos para describir la información que queremos transmitir en un fichero XML. Es un lenguaje especializado para describir nuestro lenguaje natural en código: mediante símbolos, iremos representando la estructura de lo que se quiera representar.

Un XML se compone de la declaración del XML:

```
<?xml versión= “1.0” encoding= “UTF-8”?>
```

En esta parte se declara el XML, la versión del documento y se define el *encoding* que se utilizará en el fichero. Debemos definir esta línea como nuestra primera línea de cualquier fichero XML. Los campos *versión* y *encoding* deben estructurarse en este orden para considerarse una estructura correcta. De todos modos, las declaraciones son opcionales, pero si se establece *encoding* se deberá añadir también la declaración *versión*. La versión nos servirá para indicar en qué momento se realizó el documento y seguir una evolución del estándar si se modifica el archivo en un futuro. El *encoding*, como hemos explicado en apartados anteriores, nos permitirá definir qué caracteres vamos a utilizar. Por defecto, utilizaremos UTF-8.

A continuación, añadiremos el cuerpo del XML, que es la parte más importante del fichero. Este documento adquiere una estructura de árbol, compuesto por un elemento raíz o principal dentro del cual añadiremos el resto de los elementos.

```
<raíz>
        <tronco>
                <rama1></rama1>
                <rama2></rama2>
        </tronco>
</raíz>
```

Como vemos en este ejemplo de cuerpo, podemos ver que se estructura por un elemento padre "raíz" del cual se desprenden diferentes hijos, en este caso, el elemento "tronco", o subhijos, que serían los elementos "rama". Podrá tener tantos hijos como sea necesario, pero el elemento padre no se podrá repetir. Dentro de cada etiqueta, se podrá encontrar la información de cada elemento.

Para finalizar, nuestro XML debería parecerse a algo más o menos así:

```
<?xml version="1.0" encoding="UTF-8"?>
<coches>
	<coche>
		<marca>Seat</marca>
		<modelo>Ibiza</modelo>
		<color>rojo</color>
		<matriculacion>2019</matriculacion>
	</coche>
	<coche>
		<marca>Ford</marca>
		<modelo>Focus</modelo>
		<color>gris</color>
		<matriculacion>2014</matriculacion>
	</coche>
</coches>
```

Este tipo de fichero se utiliza en muchos programas para comunicarse los unos con los otros y transportar diferentes datos entre ellos.

1.4.1. Analizadores sintácticos *(parser)* y vinculación *(binding)*. Analizadores sintácticos (*parser* DOM y SAX) y vinculación

El soporte XML en Java tiene diferentes API que nos permitirán trabajar con la información de estos archivos.

Un analizador sintáctico es básicamente un objeto que permitirá leer la información del XML y acceder a ella para extraerla. Como veremos en este tema, hay diferentes tipos que nos ofrecen diferentes ventajas frente a otros.

	DOM	SAX	JAXB
Eficiente	NO	SÍ	SÍ
Navegación bidireccional	SÍ	NO	SÍ
Manipulación del XML	SÍ	NO	SÍ
binding	NO	NO	SÍ

Tabla 7. Esquema de las características de los analizadores sintácticos.

Como vemos en esta tabla, Java soporta diferentes API para gestionar XML. A continuación, analizaremos más profundamente cada una de ellas y mostraremos algunos ejemplos.

1.4.1.1. Acceso datos DOM

Un parser es un analizador sintáctico para XML que se encarga de verificar que la estructura de ese fichero de texto es correcta.

La API DOM se caracteriza por ser un analizador basado en modelos de carga de documentos con estructuras en árbol, el cual guarda en memoria la información del XML.

DOM es una plataforma e interfaz de lenguaje estándar que permite a los programas y a los *scripts* acceder y actualizar dinámicamente el contenido, la estructura y el estilo de un documento. Se caracteriza por tener una estructura en forma de árbol.

Entre las características principales de este analizador podemos destacar que nos permitirá tener los datos en orden, navegar por ellos en ambas direcciones y disponer de una API de lectura y escritura de datos, así como también la manipulación del fichero XML.

Lo único negativo que cabe destacar es que el *parser* DOM tiene un procesado de información bastante lento, lo que provoca que consuma y ocupe mucho espacio en memoria del programa al cargar o tratar el fichero XML.

Si observamos el ejemplo, podemos ver que, para empezar, declaramos un fichero XML. Tendremos que indicarle la ruta del fichero. El fichero tendrá esta estructura con estos datos:

```
<?xml version= "1.0" encoding="UTF-8"?>
<coches>
      <coche>
            <marca>Seat</marca>
            <modelo>Ibiza</modelo>
            <color>rojo</color>
            <matriculacion>2019</matriculacion>
      </coche>
</coches>
```

Si nos fijamos, en esta ocasión le pasamos la ruta relativa del fichero. Este fichero estará en una carpeta creada dentro de nuestro proyecto, dentro de la carpeta *main/resources/xml*, aunque puede usarse la ruta absoluta, solo tendréis que cambiar esa ruta por la de vuestro fichero. A continuación, es necesario crear una nueva instancia del objeto DOM que cargará el archivo XML en la memoria.

```
try{
//Indicaremos la ruta del fichero xml
//Src es el nombre raiz de nuestro proyecto, main es la primera
capreta, resources la siguiente, dentro de xml encontraremos el fichero
//Esta es la ruta relativa.
	File arxXml = new File("src/main/resources/xml/coches.xml");

	//Creamos los objetos que nos permitiran leer el fichero
	DocumentBuilderFactory dbf =
	DocumentBuilderFactory.newInstance();
	DocumentBuilder db = dbf.newDocumentBuilder();
	//Le pasamos el XML
	Document doc = db.parse(arxXml);
	doc.getDocumentElement().normalize();
	System.out.println("Elemento raiz:" +
	doc.getDocumentElement().getNodeName());
	NodeList nodeList = doc.getElementsByTagName("coche");
	//Creamos un bucle para leer los datos del xml y los mostramos en
la consola
	for (int itr = 0; itr < nodeList.getLength(); itr++)  {
		Node node = nodeList.item(itr);

		if (node.getNodeType() == Node.ELEMENT_NODE){
			Element eElement = (Element) node;
			System.out.println("Marca: "+
			eElement.getElementsByTagName("marca").item(0).getTe
			xtContent());
			System.out.println("Modelo: "+
			eElement.getElementsByTagName("modelo").item(0).getT
			extContent());
			System.out.println("Color: "+
			eElement.getElementsByTagName("color").item(0).getTe
			xtContent());
			System.out.println("Matriculacion: "+
			eElement.getElementsByTagName("matriculacion").item(
			0).getTextContent());
		}
	}
} catch (Exception e) {
	e.printStackTrace();
}
```

En primer lugar, vamos a utilizar una nueva instancia de *DocumentBuilderFactory*, esta clase nos va a permitir obtener un analizador sintáctico que produce la jerarquía de objetos DOM a partir de documentos XML. Una vez creada la instancia de esta clase, necesitaremos definir una clase *DocumentBuilder*. Esta es necesaria para posteriormente definir el documento que nos permitirá parsear el XML.

Después crearemos una nueva instancia de la clase *Document*, que nos permitirá almacenar nuestro documento XML. La clase *Document* representa un documento XML y nos proporcionará el acceso al contenido del documento XML.

Seguidamente, necesitaremos obtener el nodo raíz a través del método *getDocumentElement()*, así obtendremos los datos de esa etiqueta.

A continuación, tenemos que detectar cuántos elementos contiene el XML, es decir, cuántos nodos hay definidos. Para ello, utilizamos el método *getElementByTagName()* al cual le pasaremos el nombre del nodo que queremos sustraer, en este caso, "coche". Para almacenar los datos obtenidos, definiremos el *NodeList*, que almacenará todos los datos encontrados haciendo la llamada al método *getElementByTagName()*. Si necesitamos acceder a todos los nodos desde el inicio del fichero, podemos llamar recursivamente a este método: *getChildElement()*.

En el ejemplo vemos también que definimos la clase *Node*, esta nos servirá para asignar todos los datos de cada elemento "coche" encontrado en el XML.

Para obtener el valor del texto, podemos usar el método *getElementByTextValue()* para buscar un nodo por su valor, y para acceder a los datos de los atributos, *getElementByTagName()* junto con el método *getAttribute()*.

Para entender mejor cómo funciona, tenéis desarrollado el ejemplo más detalladamente en el apartado correspondiente a este tema en GitLab.

1.4.1.2. Acceso datos SAX

Otro modo de acceder a los datos de un XML es con la API SAX (Simple API for XML). Esta librería se encarga de leer la información del XML línea por línea.

Esta API se caracteriza por estar basada en eventos para parsear los datos. Nos proporciona una mayor eficiencia en memoria, con acceso a los datos de bajo nivel y, en definitiva, mucho más rápida. Si la comparamos con el acceso DOM, esta es mucho más eficiente porque no carga en memoria todo el árbol del fichero. El inconveniente más destacable es que es un poco más complicada de utilizar que el resto de *parsers*, y que no ofrece navegación bidireccional.

Es capaz de encontrar el *tag* de inicio del fichero que desencadena el evento que empezará a leer los datos.

Al contrario que la API DOM, SAX no carga los ficheros en memoria, sino que lee los ficheros usando una función para informar al cliente de la estructura del documento. Por tanto, es una opción mucho más rápida de analizar el contenido de los ficheros XML que no consumirá tantos recursos de la aplicación que usemos.

Otro detalle que caracteriza a SAX es que es una interfaz que analiza los datos del fichero de forma secuencial, empezando al inicio del XML y acabando al cierre final.

Para la implementación de SAX, primero necesitaremos tener un fichero XML. Usaremos este XML que contiene información sobre coches. Tendremos que tener en nuestro proyecto o en nuestro ordenador el fichero XML.

```
<?xml version= "1.0" encoding="UTF-8"?>
<coches>
      <coche>
            <marca>Seat</marca>
            <modelo>Ibiza</modelo>
            <color>rojo</color>
            <matriculacion>2019</matriculacion>
      </coche>
</coches>
```

Para poder explicar mejor cómo estructurar una implementación de un ejemplo con SAX, vamos a desglosar el ejemplo por partes para poder explicarlo mejor.

En primer lugar, necesitamos definir la clase *SAXParserFactory* y crear una nueva instancia.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
```

Esta clase es la API encargada de proporcionar un SAX *parser*. Seguidamente, se deberá crear un SAX *parser* que se obtendrá gracias a la llamada del método *newSaxParser()*.

Como vemos en el ejemplo, para poder parsear el XML deberemos llamar al método *parse()*, al que se necesita pasarle por parámetro la ruta del fichero XML y un *handler*. Lo que hará es llamar a una clase auxiliar de apoyo que veremos cómo se crea a continuación.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();

SaxHelper handler = new SaxHelper();
saxParser.parse("src/main/resources/xml/coches.xml", handler);
```

Un ***handler*** es una clase auxiliar que servirá para realizar los diferentes pasos de extracción de datos del XML. Deberemos crear una clase auxiliar llamada *SaxHelper* que

extenderá de *DefaultHandler* y que nos proporcionará la implementación por defecto de los métodos necesarios para realizar el *parser*.

Para entender mejor en qué consiste, vamos a ver su implementación:

```
public class SaxHelper extends DefaultHandler{

	…
}
```

La clase se llamará en este caso "SaxHelper", pero podemos asignarle el nombre que más nos convenga. Este método se llamará cuando se encuentre el principio de un elemento, este deberá extender siempre de la clase *DefaultHandler*. De este modo, nos podremos beneficiar de los métodos por defecto. Si no se implementa así, no nos funcionará el *parser* con SAX.

```
public class SaxHelper extends DefaultHandler{

	boolean esMarca = false;
	boolean esModelo = false;
	boolean esColor = false;
	boolean esMatriculacion = false;
}
```

A continuación, debemos crear tantas variables como atributos tiene nuestro XML. En nuestro caso, tenemos cuatro y hemos creado cuatro booleanos con sus nombres asociados. Estas variables nos servirán para saber si el elemento que estamos comprobando corresponde a cada atributo que queremos encontrar. Por tanto, tendremos uno para marca, modelo, color y matriculación.

Nuestra clase auxiliar, para funcionar correctamente, tendrá que implementar un método *startElement*, el cual se encargará de buscar los diferentes elementos y atributos de nuestro XML. Este método se llamará cuando se encuentre el principio de un elemento.

```
public void startElement(String uri, String localName, String elementos,
Attributes atributos) throws SAXException {
	System.out.println("Inicio del elemento :" + elementos);

	switch (elementos) {
		case "marca":
			esMarca = true;
			break;
		case "modelo":
			esModelo = true;
			break;
		case "color":
			esColor = true;
			break;
		case "matriculacion":
			esMatriculacion = true;
			break;
		default:
			break;
		}
}
```

Si nos fijamos, el método recibe cuatro parámetros:

- URI: contendrá el *namespace* del XML si lo tiene. En este caso estará vacío.
- *localName:* puede ir vacío.

> Un ***namespace*** es una manera de indicar al XML que un elemento es diferente de otro elemento con el mismo nombre.

- Elementos: contendrá el nombre del elemento que se acaba de encontrar.
- Atributos: contendrá los atributos de cada elemento.

Si nos fijamos, el método *startElement* se encarga de ir comprobando que el *string* "elementos" contenga uno de nuestros elementos del XML. Cuando encuentre uno de ellos, establecerá el valor *true* a la variable que corresponda según la etiqueta.

A continuación, deberemos implementar el método *characters()*. Este método será llamado cuando se encuentra texto en el XML. Es un método que recibe por parámetro:

- Un *array* de tipo *char* que contendrá todos los caracteres de nuestro XML.

- Un *int* inicio que contendrá un *int* que indicará la posición en la que tiene que empezar a leer el *array char* anterior.
- Un *int lenght* que indicará el número de caracteres que tenemos que usar del *array* de caracteres.

```
public void characters(char ch[], int inicio, int length) throws
SAXException {
	if (esMarca) {
		System.out.println("Marca: " + new String(ch, inicio,
length));
		esMarca = false;
		return;
	}

	if (esModelo) {
		System.out.println("Modelo: " + new String(ch, inicio,
length));
		esModelo = false;
		return;
	}

	if (esColor) {
		System.out.println("Color: " + new String(ch, inicio,
length));
		esColor = false;
		return;
	}

	if (esMatriculacion) {
		System.out.println("Matriculacion: " + new String(ch,
inicio, length));
		esMatriculacion = false;
		return;
	}
}
```

El método se encargará de comprobar qué elemento debe imprimir por consola, ya que en el método anterior hemos verificado qué elemento estábamos comprobando. Si la propiedad booleana correspondiente a ese elemento del XML es *true*, imprimirá por consola el elemento y el valor correspondiente, y volverá a setear a *false*. Esto se hace para indicar que ya no está en uso ese elemento.

Para finalizar con este *helper*, necesitaremos implementar el método *endElement()*. El método se llama cuando encuentra el final de un elemento. En este ejemplo, solo se

```
public void endElement(String uri, String localName, String
elementos) throws SAXException {
	System.out.println("Fin del elemento: " + elementos);
}
```

imprimirá por consola el valor del elemento e indicará que hemos terminado con este bloque del XML.

Este método recibe por parámetro un *string* con la URI, otro con *localName* y otro con el elemento XML finalizado. En este caso, solo tendrá valor el *string* "elementos" porque nuestro XML es más sencillo y estos campos están vacíos. Cabe decir que solo se rellenarán cuando el XML contenga esa información. Como vemos en el ejemplo, los objetos para acceder a los datos de un fichero son con *SaxParserFactory* y *SaxParser*.

Este es un ejemplo muy simplificado para observar el funcionamiento básico y los objetos. Es recomendable consultar GitLab para ver el ejemplo implementado, ya que la implementación de SAX es bastante complicada.

1.4.1.3. Acceso a datos JAXB *(binding)*

Cuando trabajamos con un lenguaje de programación como Java, las posibilidades para jugar con el lenguaje son muy extensas. En este caso, analizaremos cómo podemos tratar los XML y a qué otros formatos podemos convertir ese archivo.

Una de las posibilidades más interesantes que nos ofrece Java es la posibilidad de convertir un XML en un objeto Java. La librería Java JAXB (Java XML API Binding) nos ofrece esa posibilidad. Está incluida dentro de la JDK de Java, por lo que no es necesario importar librerías.

El *data binding* es el concepto para definir la voluntad de transformar un fichero XML (o cualquier otro fichero) en un objeto Java.

Algunas de las características principales de esta API son que nos permite navegar por el documento en ambas direcciones, es más eficiente que DOM y permite la conversión de tipos XML a Java. Además, permite la manipulación del fichero XML. Trabaja de manera muy simple la carga, creación y manipulación de ficheros XML.

Como aspecto negativo, cabe destacar que no permite tratar un fichero XML si este no es válido, por tanto, tenemos que validarlo antes de poder transformarlo a un objeto Java.

```
String xml = "<?xml version= \"1.0\" encoding=\"UTF-8\"?>"
+ "<coches><coche><marca>Seat</marca><modelo>Ibiza</modelo>" +
"<color>rojo</color><matriculacion>2019</matriculacion>"
+ "</coche></coches>";

JAXBContext jaxbContext;

try {
      jaxbContext = JAXBContext.newInstance(Coche.class);
      // Este objeto se encargará de la transformación a objeto
      Java que le indiquemos.
      Unmarshaller jaxbUnmarshaller =
jaxbContext.createUnmarshaller();
      Coche coche = (Coche) jaxbUnmarshaller.unmarshal(new
StringReader(xml));
      System.out.println(coche);
} catch (JAXBException e) {
      e.printStackTrace();
}
```

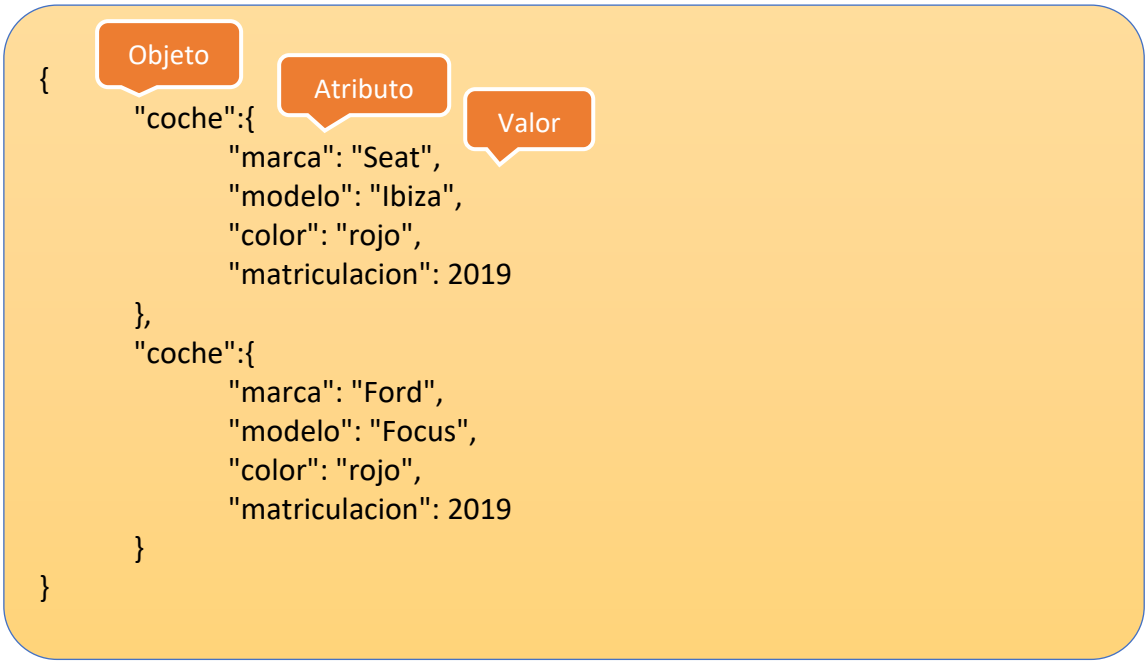
Tal y como podemos ver en el ejemplo, la API JAXB se encarga de coger los datos del XML y convertirlos en el objeto que le indiquemos. El objeto debe tener los mismos atributos que el XML, si no, se lanzará la excepción.

1.4.2. Librerías para la conversión de documentos XML a otros formatos

En Java existen una infinidad de librerías que podemos añadir a nuestro proyecto para poder transformar nuestros XML a cualquier otro formato. En este apartado, explicaremos cómo transformar nuestros ficheros XML a un formato JSON.

JSON: siglas para Javascript Object Notation. Es un tipo de archivo de formato de texto derivado de Javascript, bastante ligero y que almacena información estructurada. Es fácil de interpretar y generar, y se utiliza para transferir información entre un cliente y un servidor.

Los archivos JSON siguen una estructura basada en definición de objetos, asignando un atributo y un valor. Un fichero JSON es capaz de definir seis tipos de valores: *string*, número, objeto, *arrays*, *true*, *false* o *null*. Veamos un ejemplo:



Para definir el JSON, se abren y cierran corchetes. Los objetos se declaran entre comillas y los diferentes objetos se separan con una coma. El nombre y el valor de cada pareja van separados entre dos puntos. Cada objeto se considera un *string;* en cambio, los valores de los atributos pueden ser de cada tipo permitido nombrado en el parágrafo anterior.

XML a JSON

```
public static String XML_PRUEBA =
"<coche><id>1</id><modelo>Ibiza</modelo><marca>seat</marca></coc
he>";
try {
      //Creamos el objeto que nos ayudara a convertir el XML en
JSON
      JSONObject json = XML.toJSONObject(XML_PRUEBA);
      //Identamos el json, le damos formato
      String jsonFormatado = json.toString();
      System.out.println(jsonFormatado);
} catch (JSONException je) {
      System.out.println(je.toString());
}
```

Para esta conversión tenemos que añadir la librería Jackson a nuestro proyecto.

Esta librería nos permitirá realizar la transformación de XML a JSON en muy pocas líneas de código.

Se puede descargar en estos enlaces:

https://repo1.maven.org/maven2/org/json/json/20190722/

https://jar-download.com/artifacts/org.json

Este sería un ejemplo sencillo de la implementación de esta librería: en primer lugar, tenemos que definir un XML, en el ejemplo tenemos un modelo sencillo de XML. A continuación, definiremos un nuevo objeto usando la clase *JSONObject* que se rellenará con la llamada al método *XML.toJSONObject()*. Este método nos permitirá transformar cualquier XML a JSON, pasándole el XML como *string*. Una vez transformado a objeto JSON, podemos pasarlo a *string* usando el método *toString()* y ya tendríamos la transformación realizada. Tenemos que tener en cuenta que nuestro XML debe estar bien estructurado y tiene que estar validado y sin ningún error. Si hay algún error, no se hará la transformación correctamente y saltará un error que lo controlaremos en el *catch*.

1.5. Excepciones: detección y tratamiento

El tratamiento de las excepciones es un proceso muy importante en la creación de un programa. Es el mecanismo que nos permite controlar un proceso anómalo dentro de nuestro programa y gestionar qué ocurre cuando se produce un error.

Una excepción es un evento que se produce durante la ejecución de un programa que interrumpe el flujo normal de ejecución del código a causa de un error, ya sea controlado o inesperado.

Para poder gestionar los errores, en Java utilizamos lo que se llama un *try-catch*. Un *try-catch* es un bloque de código que se añadirá a nuestro programa siempre que queramos controlar una parte del código que vayamos a desarrollar. El código se ejecutará en orden e irá siguiendo hasta que se produzca un error, en ese momento entrará al *catch*. Podemos definir más de un *catch* en un mismo *try*. Cuando se produzca el error, entrará en el primer bloque *catch* que coincida con el tipo de excepción y solo podrá entrar en uno de los bloques *catch*. Los tipos de excepción más específicos deben aparecer primero en la estructura, seguidos de los tipos de excepción más generales.

El *try* es lo que primero se ejecuta: se prueba la acción que se intenta realizar y, si se produce un error, el *catch* recoge la excepción y, según lo que indiquemos, lanza un mensaje con la descripción de lo que ha ocurrido. Al lanzar una excepción, el proceso normal del programa se interrumpirá.

Lo mejor para controlar los errores es tener en cuenta todas las posibilidades de error y controlarlas. Se debe utilizar un *try-catch* siempre que sea necesario.

```
try{
//Parte donde se ejecuta el código que puede dar excecion
        File prueba = new File("c\\:prueba.txt");
} catch(NoSuchFileException e){
        System.out.println("No se ha encontrado el archivo" +e);
}catch(IOExcepction ex){
//Parte donde se gestiona la excepción
        throw ex;
}
finally{
        //Bloque de código que se ejecutará siempre, aunque se lanze una //excepcion
}
```

Como vemos en el ejemplo, antes de realizar la acción debemos envolverla con un *try-catch*. Dentro del bloque del *try*, añadiremos la parte del código que queremos controlar si se produce un error. Si no se produce el error, el programa continuará la ejecución y entrará dentro del bloque *finally*. Este bloque se ejecutará siempre, aunque no se produzca un error, y es especialmente útil para cerrar el proceso o ejecutar algún tipo de acción necesaria. En el caso de que se produzca un error, entrará en el *catch* y realizará las acciones dentro de ese apartado y lanzará un error. Para lanzar el error, lo realizaremos mediante la palabra "throw" seguida de la excepción definida, en este caso, "ex". Cada excepción tiene diferentes métodos para dar detalle de lo que está ocurriendo. Siempre que forme parte de la familia *Throwable*, tendremos estas opciones.

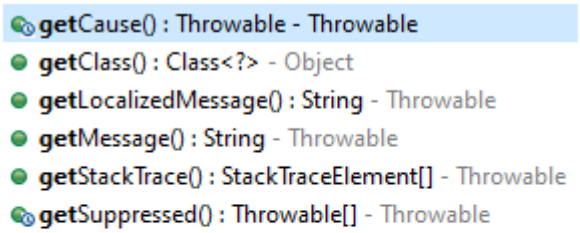


Ilustración 8. Métodos de la clase IOException.

Para saber qué métodos contiene, podemos dirigirnos a la página web de la documentación de Java, donde obtendremos toda la información.

https://docs.oracle.com/javase/8/docs/

Cuando se produce un error y hemos definido un bloque *finally*, también se ejecutará. Como hemos explicado en el apartado anterior, debemos definir las excepciones más específicas primero y dejar para lo último las más genéricas. Podemos definir cuantos *catch* sean necesarios.

Cuando programemos en Java, podremos encontrar dos tipos de excepciones: las excepciones controladas y las excepciones no controladas.

Excepciones controladas *(checked exceptions)*

Entendemos por **excepciones controladas** todas aquellas que representan errores fuera del control del programa. Estas excepciones se controlan en tiempo de ejecución.

Usaremos este tipo de excepción siempre que se espere que el programa puede recuperarse después de lanzarse la excepción. Son de uso obligatorio en cualquier gestión de ficheros.

Este tipo de excepciones son subclases de la clase *Exception*.

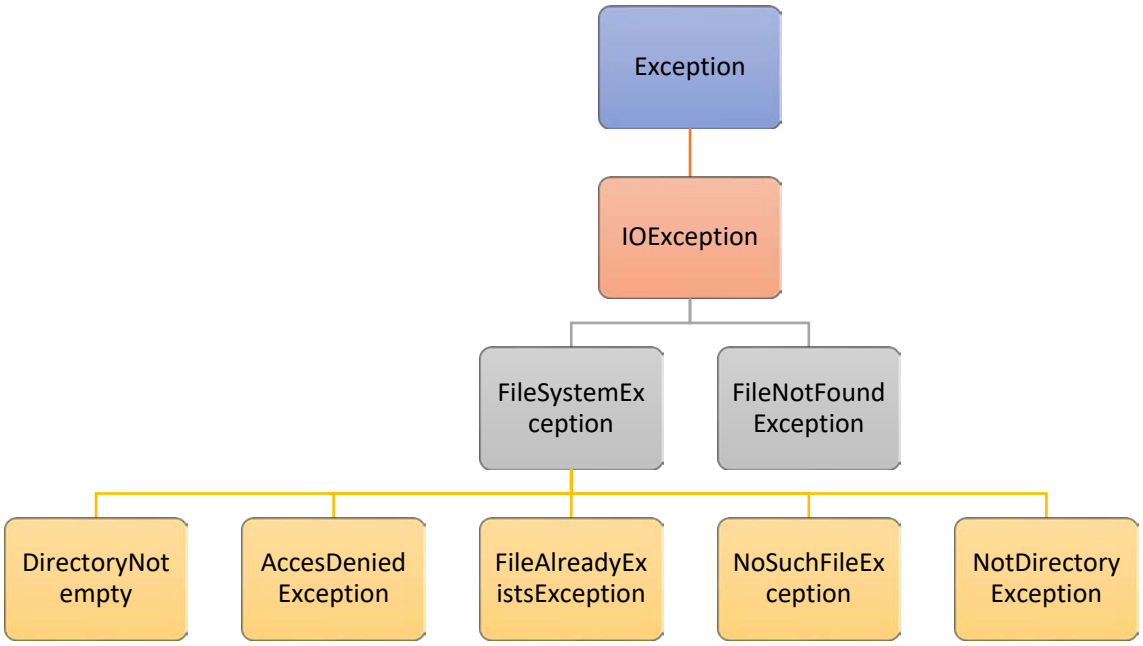


Ilustración 9. Gráfico con la jerarquía de excepciones más comunes en gestión de ficheros.

- *IOexception:* es una clase de error genérico, la podemos usar siempre que queramos controlar todo tipo de excepción sin saber exactamente cuál se podrá lanzar. Subclase de *Exception*.
- *FileSystemException:* es una clase que lanza errores cuando una operación con ficheros falla en un fichero o dos.
- *DirectoryNotEmptyException:* nos indicará que la carpeta no está vacía.
- *FileNotFoundException:* nos servirá para indicar que no se ha encontrado el fichero.

- *AccesDeniedException:* esta excepción es útil para controlar si el acceso al fichero está permitido.
- *FileAlreadyExistsException:* cuando se crea un fichero y ya existe se lanza esta excepción.
- *NoSuchFileException:* será útil para controlar si existe el fichero al cual queremos acceder.
- *NotDirectoryException:* para controlar si existe la carpeta a la cual queremos acceder o crear.

Hay dos maneras de controlar las excepciones:

```
private static void excepcionControladaConThrows() throws
FileNotFoundException {
    File ej = new File("fichero.txt");
    FileInputStream stream = new FileInputStream(ej);
}

private static void metodoConTryCatch() {
    File ej = new File("ejemplo.txt");
    try {
        FileInputStream stream = new FileInputStream(ej);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Como podemos ver en el ejemplo, hay dos tipos de excepción: la que se pone en la declaración del método con un *throws* o con un *try-catch* dentro de la implementación del método. Hay que tener en cuenta que, cuando declaramos el *throws* en el método, quien recibe la función deberá envolver esa llamada dentro de un *try-catch*, es decir, quien haga la llamada al método *excepcionControladaConThrows()* deberá realizarla con un *try-catch*. Algo más o menos así:

```
private static void ejemploLlamada() {

    try {
        excepcionControladaConThrows();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Excepciones no controladas *(unchecked exceptions)*

Las excepciones **no controladas** son aquellas que reflejan un error dentro de la lógica del programa que estamos desarrollando.

Este tipo de excepciones no se comprueban en tiempo de compilación, por tanto, no es necesario englobar este tipo de excepción en un *throws* o un *try-catch*.

En la gestión de ficheros, este tipo de excepción no será la más usual, pero sí que se tiene que tener en cuenta, porque puede que se reproduzca en algún momento si alguno de los parámetros que usemos esta vacío o *null*.

Los más usuales serán:

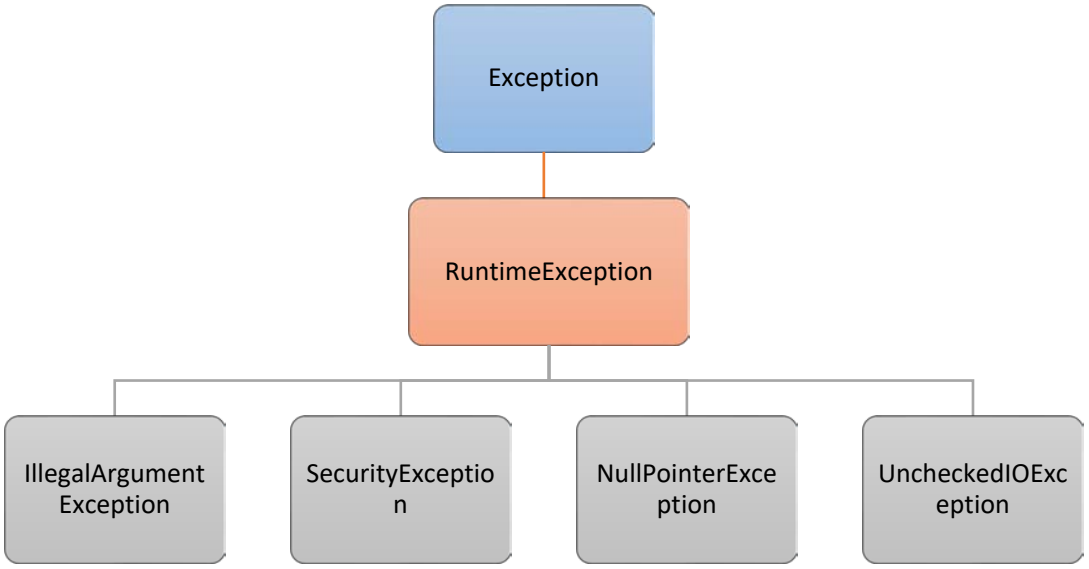


Ilustración 10. Gráfico con la jerarquía de excepciones no controladas más comunes en gestión de ficheros.

- *RuntimeException:* se lanza cuando se produce un error con la máquina virtual de Java.
- *NullPointerException:* se lanzará siempre que uno de los campos que se le pasen al programa esté vacío.
- *IllegalArgumentException:* se lanzará esta excepción cuando uno de los argumentos que se le pasen a un método no sea válido.
- *SecurityException:* cuando se produce algún problema de seguridad.
- *UncheckedIOException:* cuando se produce una excepción no controlada.

1.6. Pruebas y documentación de las aplicaciones desarrolladas

Una de las partes más importantes de la programación y en la que seguramente muchas veces nadie piensa es la documentación. Dejar el código bien documentado y comentado es una tarea igualmente importante y que en nuestra trayectoria laboral nos será de gran utilidad.

La manera más conocida de documentar el lenguaje Java es JavaDoc. Se trata de una utilidad de Oracle que permite documentar clases de Java.

En primer lugar, lo más común es usar comentarios. Los comentarios se pueden escribir de tres maneras:

```
//Comentario en una línea simple
/*
*Comentario en más de una línea
*/
/**
*Comentario con JavaDoc
*/
```

La estructura utilizada en JavaDoc es muy similar a la que se utiliza normalmente para comentarios, pero en la API se añade un asterisco extra al empezar, que es el característico para definirlo.

La gran peculiaridad de esta API es que permite añadir *tags* HTML dentro de los comentarios. Eso es debido a que JavaDoc, a través de los comentarios, genera un documento que servirá como documentación del proyecto.

Los comentarios para documentar, normalmente, se pueden poner en el código, encima de cualquier clase, en los métodos o atributos que queramos documentar.

Cuando se añadan comentarios, es necesario tener en cuenta que se debe describir cuál es la función de lo que se comenta. Por ejemplo, si comentamos una clase, haremos una breve explicación de lo que se encarga.

```
/**
 * Aquí pondremos un resumen de la finalidad de esta clase
 * Ej: Clase para documentar el funcionamiento de JavaDoc
 *
 * @author Ilerna
 *
 */
public class JavaDoc {

    /**
     * Atributo nombre de la clase javaDoc
     */
    private String nombre;

    /**
     * <p>Ejemplo de Javadoc con tags html .
     * <a href="http://www.ilerna.es">Web ilerna</a>
     * </p>
     * @param ejemplo String que pasamos por parametro
     * @return String con el resultado del metodo
     * @see <a href="http://www.ilerna.es">Ejemplos</a>
     * @since 1.0
     */
    public String ejemploMetodoJavadoc(String ejemplo) {
        // Comentario con explicación de lo que se realiza en el
    método
        return “OK”;
    }

}
```

Tags más importantes:

TAG	DESCRIPCIÓN
@author	Sirve para poner el autor del desarrollo.
@deprecated	Indica que el método o clase es obsoleto (propio de versiones anteriores) y que no se recomienda su uso.
@param	Se usara para definir un parámetro de un método, es requerido para todos los parámetros del método.
@return	Se usa para indicar qué es lo que devuelve el método, no se usa para los métodos *void*.
@see	Asocia con otro método o clase.
@version	Se usa para definir la versión del método o la clase

Tabla 8. Etiquetas más importantes para JavaDoc.

Para empezar, vamos a ver un ejemplo de cómo implementar JavaDoc en una clase.

Todos estos comentarios nos servirán para crear la documentación. Para generarla, podemos hacerlo desde la línea de comandos ejecutando este comando:

```
javadoc –d doc src\
```

También podremos hacerlo a través de nuestro IDE. Si accedemos a *Project > Generate JavaDoc,* nos aparecerá una ventana que nos permitirá elegir dónde queremos guardar nuestra documentación. Y ya lo tendremos creado.

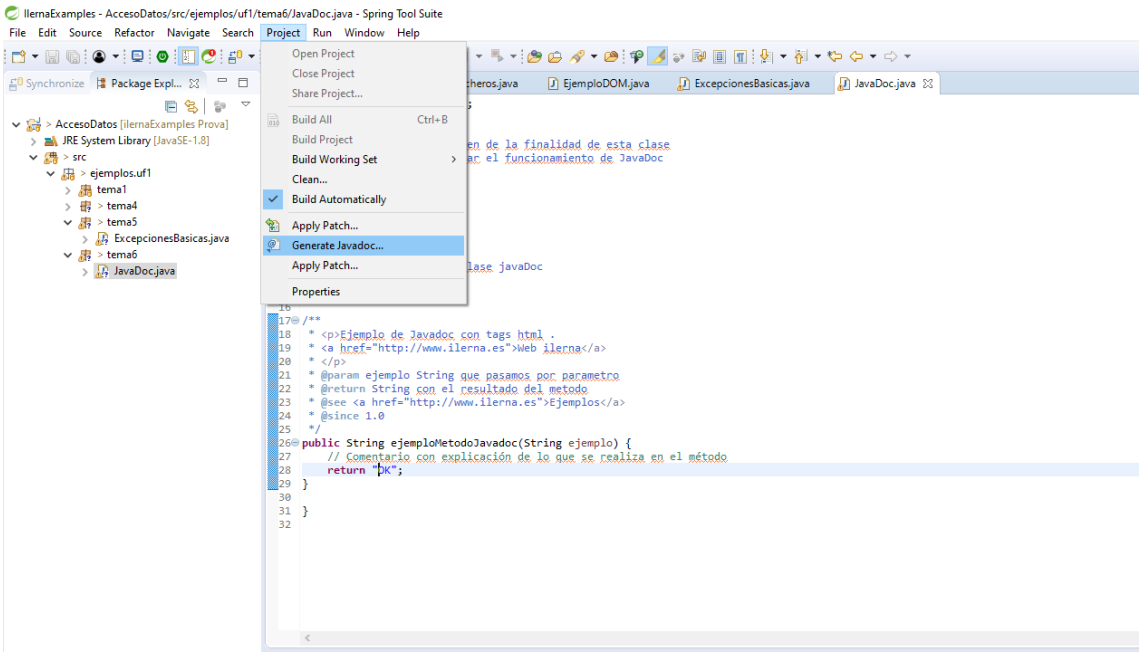


Ilustración 11. Captura con las indicaciones para generar un JavaDoc.

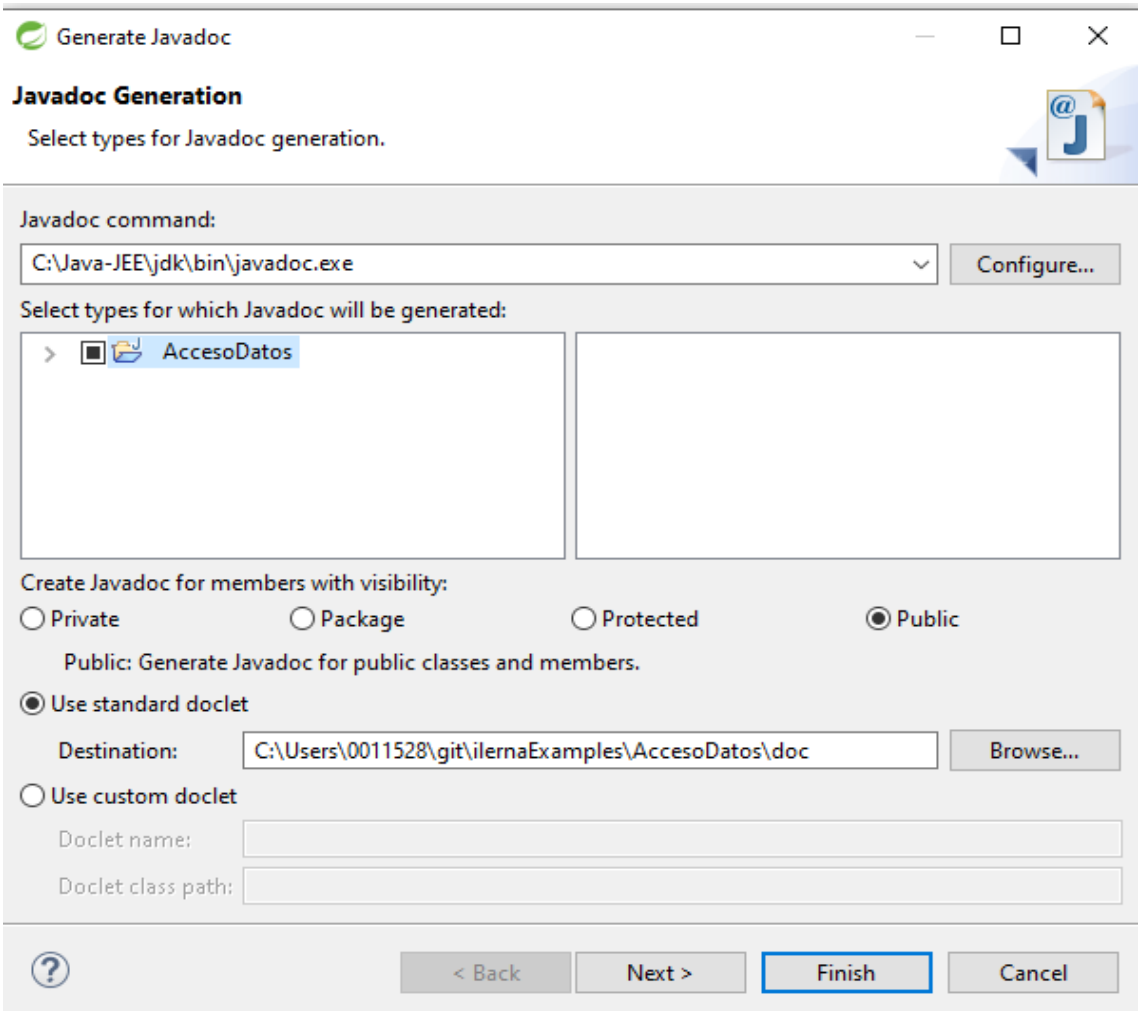


Ilustración 12. Ventana para indicar dónde queremos guardar nuestro JavaDoc.

A continuación, pondremos en práctica lo aprendido en esta lección y realizaremos pruebas con JUnit. Para los que nunca hayan escuchado hablar de ello, JUnit es un *framework* muy conocido para hacer pruebas de código.

Antes de empezar, detallaremos las etiquetas y métodos más importantes que usaremos para construir nuestros test de pruebas.

TAG	DESCRIPCIÓN
@Test	Identifica un método como un método test.
@Before	Indica qué se ejecuta antes de cada test. Se usa para preparar el entorno del test antes de ejecutarlo.
@After	Indica qué se ejecutará después de cada test. Se usa para limpiar el entorno después de la ejecución de cada test.
@BeforeClass	Se ejecuta solo una vez, antes de que se ejecuten todos los test. Se usa, por ejemplo, para conectar la base de datos.
@AfterClass	Se ejecuta solo una vez después de ejecutar todos los test. Se usa para limpiar el entorno.
@Ignore or @Ignore ("why disabled")	Marca ese test como deshabilitado.
@Rule	Establece una regla, añade funcionalidades extra a nuestro test.

Tabla 9. Relación de etiquetas más usadas para los test con JUnit.

MÉTODO	DESCRIPCIÓN
fail([message])	Se usa para hacer que el método falle y descubrir qué partes del código no se contemplan. Se suele hacer antes de desarrollar. El parámetro *message* es opcional.

assertTrue([message,]boolean condition)	Comprueba si la condición es *true*.
assertFalse([message,]boolean condition)	Comprueba si la condición es *false*.
assertFalse([message,]boolean condition)	Hace test de dos valores para comprobar si son el mismo.
assertEquals([message,] expected,actual, tolerance)	Comprueba si un *float* o *double* son iguales.
assertNull([message,] object)	Comprueba si un objeto es *null*.
assertNotNull([message,] object)	Comprueba si el objeto no es *null*.
assertSame([message,] expected, actual)	Comprueba si las dos variables hacen referencia al mismo objeto.
assertNotSame([message,] expected, actual)	Comprueba si las dos variables no hacen referencia al mismo objeto.

Tabla 10. Relación de métodos más importantes para la ejecución de JUnit.

Para poder realizar los ejemplos y las prácticas, deberemos preparar Eclipse para que podamos desarrollar los test.

En primer lugar, nos dirigiremos a la siguiente web para descargar la librería de JUnit:

https://mvnrepository.com/artifact/org.junit/junit5-api/5.0.0-ALPHA

Después debemos añadir esta librería a nuestro proyecto en Eclipse. Si no tenéis ningún proyecto creado, os aconsejo que os descarguéis el proyecto con los ejemplos de este tema. Os será útil para realizar la práctica (el enlace está al principio del tema 1).

Para ello, tenemos que dar botón derecho al proyecto e ir a *Properties*. A continuación, dirigirnos *a Java Build Path* y a librerías, tal y como se muestra en la imagen.

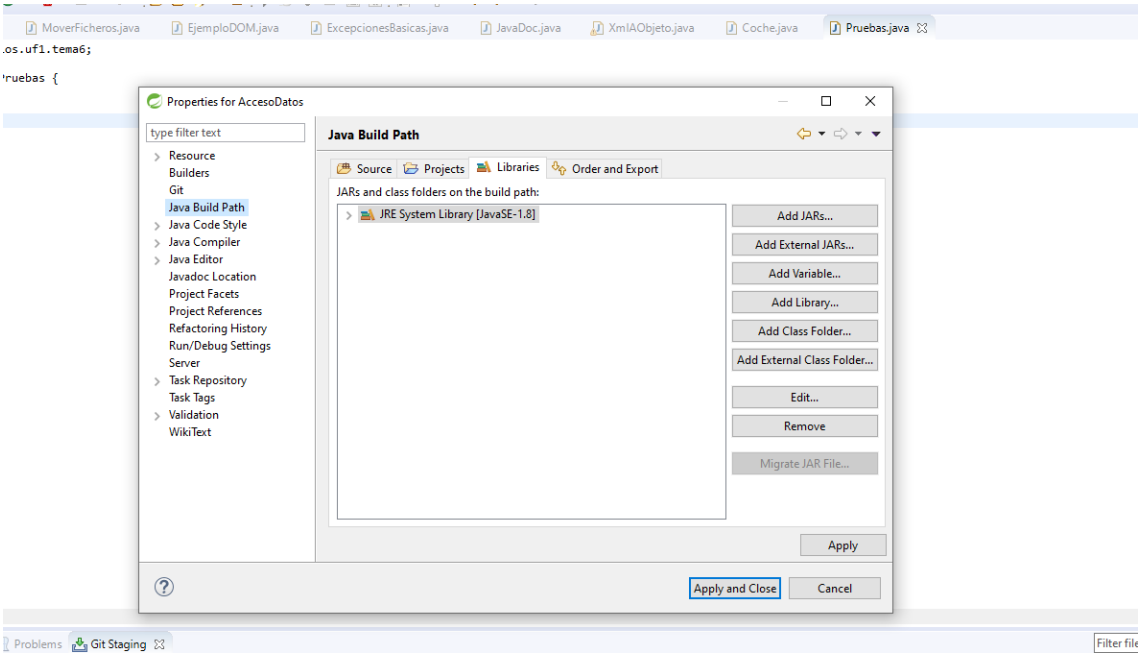


Ilustración 13. Captura con las instrucciones para añadir la librería de JUnit.

Luego clicamos el botón *Add Library* y nos aparecerá este *pop up*. Tendremos que elegir *JUnit*.

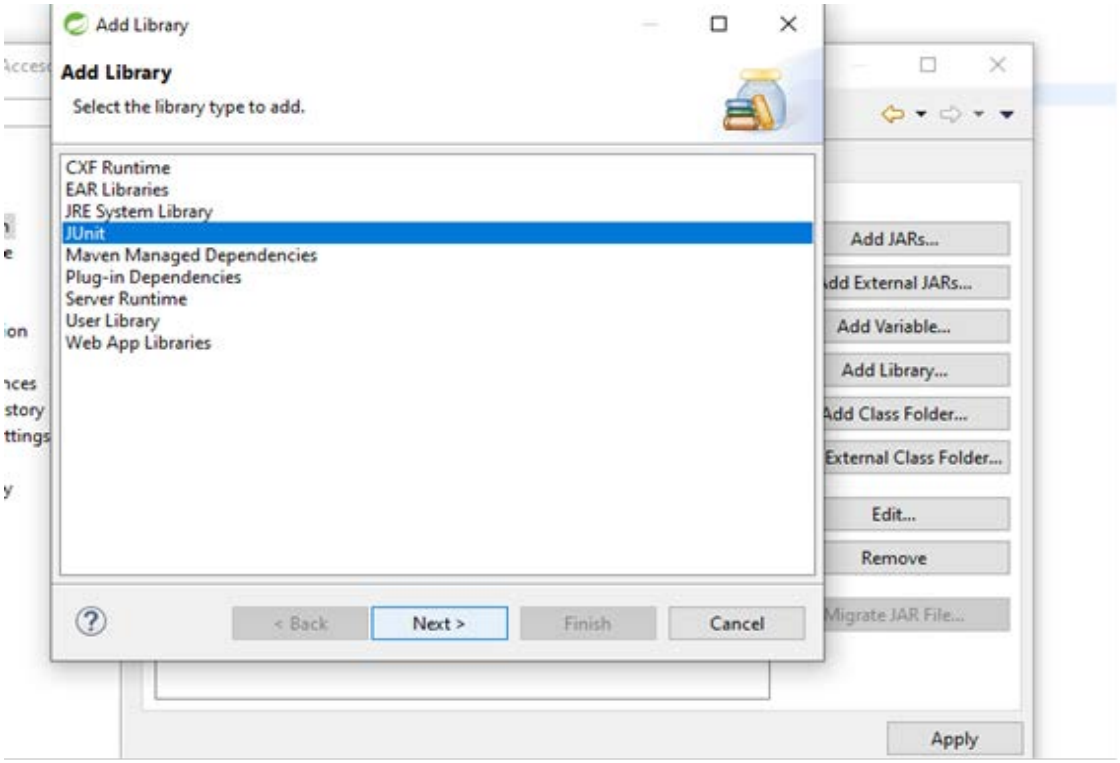


Ilustración 14. Captura con las instrucciones para añadir la librería de JUnit, segunda parte.

Seguidamente, nos saldrá otra pantalla que nos permitirá elegir la versión de JUnit, que en esta ocasión será la 5, que es la compatible con la versión 8 de Java. Es importante tener el .jar copiado en la carpeta de *plugins* del proyecto, si no, Eclipse no lo va a encontrar.

Cuando realizamos un test con JUnit, debemos crear una carpeta en el proyecto específica solo para el test. Dentro de la carpeta "src" de nuestro proyecto, crearemos una nueva carpeta llamada "test" y dentro de esta otra llamada "Java". Aquí podremos ir creando las clases para nuestros test.

Para practicar algunos de los ejemplos que hemos realizado durante este tema, crearemos diferentes test para diferentes acciones con ficheros.

```
//Etiqueta que indicara que este método es un test.
@Test
public void testCreacion() throws IOException{
        File archivo = tempFolder.newFile("prueba.txt");
        assertTrue(archivo.exists());
}
```

En este ejemplo, podemos ver la realización de un método para testear si se puede crear correctamente un fichero. Este es un ejemplo sencillo que ilustra la facilidad y la utilidad de realizar test en nuestros proyectos.

Para ejecutar el test, solo debemos dar al botón derecho encima de la clase y darle a *Run As > JUnit Test*.

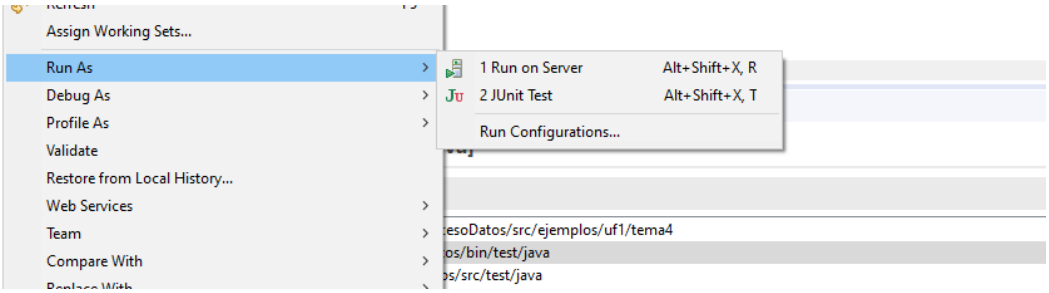


Ilustración 15. Captura ejemplo para ejecutar un test de JUnit.

En GitLab encontraréis más ejemplos de cómo realizar estos test.

2. Gestión de conectores (desarrollo de aplicaciones que gestionan información en bases de datos relacionales)

En este tema introduciremos todos los conceptos relacionados con bases de datos y aprenderemos a realizar todo tipo de conexiones y consultas a bases de datos.

Una **base de datos** es un conjunto de información almacenada en un servidor o un disco que permite guardar información de cualquier tipo para poder recuperarla y usarla en una aplicación cuando sea necesario.

Cada base de datos se compone por un grupo de tablas. Cada tabla se compone por una o más columnas que definirán el nombre de cada elemento que guardar. Cada registro será una fila en esta tabla, que corresponderá a uno de los datos que guardar.

En programación, es muy común usar bases de datos relacionales.

Una **base de datos relacional** es un tipo de base de datos en la cual las tablas se relacionan entre sí.

En una base de datos relacional, cada fila de una tabla contiene un registro único llamado ID que se definirá como una clave única que ninguna fila más de la tabla puede compartir. Ese identificador es el que nos facilitará la relación entre una tabla y otra.

Este tipo de base de datos es especialmente útil para las aplicaciones que tienen que gestionar una gran cantidad de datos que administrar. Además, facilitan la coherencia de datos, ya que permiten varias instancias de base de datos.

2.1. Gestores de bases de datos embebidos e independientes

En este apartado, profundizaremos en los conceptos básicos de la base de datos embebida y explicaremos las BBDD más comunes de este tipo.

Cuando desarrollamos una aplicación, en muchos casos, es necesario el uso de bases de datos para almacenar la información que se gestiona. Normalmente, se usan bases de datos que guardan los registros en el disco (cuando trabajamos en una aplicación en local) o en un servidor, pero existen otras alternativas. Aquí entran las bases de datos embebidas.

Las **bases de datos embebidas** son aquellas bases de datos que son incorporadas dentro del *software* del programa por el desarrollador, de manera que es invisible para el usuario.

Este tipo de BBDD no necesita ninguna interfaz externa para mantener la base de datos. La característica principal es que se ejecuta al arrancar la aplicación, es decir, cuando la ejecutamos en el servidor. Se recomienda su uso en aplicaciones que no requieran almacenar datos por parte de los usuarios o para probar aplicaciones en el entorno de desarrollo. Se adapta perfectamente a las aplicaciones que solo requieren un repositorio para mantener su propia transacción sin la intervención del usuario final.

Sin embargo, es necesario recalcar que este tipo de base de datos ofrece una falta de cohesión, y se puede convertir en una opción muy difícil de mantener al no estar registrada en un disco.

En las siguientes secciones veremos algunas de las bases de datos en memoria más utilizadas para el entorno Java y la configuración necesaria para cada una de ellas.

Podemos encontrar diferentes opciones de BBDD embebidas. Aquí detallamos las más conocidas:

- **SQLite**: es un gestor de BBDD muy usado en lenguajes como Java, PHP, VisualBasic o Perl. Se caracteriza por ser muy ligero, multiplataforma, no requiere instalación y ofrece un buen rendimiento. Como curiosidad principal, la base de datos se guarda como un fichero guardado en la aplicación.
- **H2**: es un gestor de código abierto y se caracteriza por soportar SQL para gestionar la base de datos. También se trata de una BBDD multiplataforma, tiene un tamaño reducido y es una de las más rápidas.
- **Apache Derby**: usa el modelo de base de datos relacional. Se caracteriza por tener un tamaño muy reducido y es una opción muy fácil de instalar, usar e implementar.
- **Firebird**: también es un gestor de base de datos de código abierto, multiplataforma y que es soportado por diferentes lenguajes.

- **Oracle Embedded**: se caracteriza por su fácil instalación y su configuración automática. Su rendimiento es de los más eficaces, se adapta a las aplicaciones móviles gracias a que consume muy pocos recursos en este tipo de dispositivos.
- **HSQLDB**: es otra opción disponible que se caracteriza por ser muy rápida en la gestión de consultas, es de código libre y permite gestión de bases de datos relacionales.

2.2. El desfase objeto-relacional

Una de las características principales de la programación en el lenguaje Java es que se trata de un lenguaje orientado a objetos, lo que será clave para diseñar nuestra futura aplicación.

La aplicación que desarrollemos requerirá una conexión a una base de datos para poder persistir los datos que la aplicación gestione. Para ello, necesitaremos diseñar una base de datos relacional, y es donde nos encontraremos los primeros problemas. Podemos entender que:

La **programación orientada a objetos** es un paradigma de programación que permite desarrollar aplicaciones complejas con una estructura más clara y entendible, en la que se organiza el código en clases, en las cuales predominan, por su importancia, los objetos.

En otras palabras, la programación orientada a objetos se caracteriza por gestionar entidades y realizar asociaciones entre ellos. Por otro lado, las bases de datos utilizan otro tipo de nexo para entrelazarse, que no es otro que identificadores. Cada tabla se puede relacionar con otra con una columna de un identificador de otra. Es por ese motivo, y por la diferencia de relaciones entre un lenguaje orientado a objetos y la base de datos, que se produce este tipo de desfase.

El **desfase objeto-relacional** es aquel que surge cuando se desarrollan aplicaciones orientadas a objetos usando una base de datos de tipo relacional.

Para entender mejor el concepto, lo que ocurre en nuestra aplicación es que tendremos algo más o menos así:

```
public class Estudiante {
      private int id;
      private String dni;
      private String nombre;
      private String apellido;
      private int edad;

//constructor
// getters y setters
}
```

Un objeto llamado "Estudiante", con diferentes atributos. En cambio, en la base de datos, cuando creamos la tabla, normalmente creamos campos que se asemejen a los definidos en el objeto de la clase.

Como podemos ver, no están definidos del mismo modo que en el objeto, sino con un equivalente, y realizamos el mapeo de manera manual. Es decir, cada vez que tengamos que añadir o modificar registros, se tendrán que hacer de manera manual a través de consultas de base de datos. Además, los tipos de atributos en un objeto y en una tabla no se definen con el mismo tipo, sino con equivalentes.

```
CREATE TABLE estudiante (
  id INT PRIMARY KEY AUTO_INCREMENT;
  dni VARCHAR(50) NOT NULL,
  nombre VARCHAR(50) NOT NULL,
  apellido VARCHAR(100),
  edad INT DEFAULT 10;
);
```

2.3. Conexión a bases de datos

En este apartado del tema, aprenderemos cómo conectar nuestra aplicación de Java a una base de datos, cómo realizar diferentes consultas y las acciones básicas para tratar los datos.

2.3.1. Protocolos de acceso a bases de datos. Conectores

Cuando tenemos una aplicación creada en Java, en muchos casos es necesario guardar datos en una base de datos. Para ello, es necesario establecer una conexión entre la aplicación y la base de datos.

En ese caso, es necesario utilizar una interfaz para acceder a la base de datos desde Java con la API JDBC (Java Database Conectivity). A través de ella, se establece la conexión a la BBDD, que nos permitirá realizar consultas, actualizar los datos y recibir resultados de las consultas. Permite realizar operaciones con lenguaje SQL independientemente de la instancia de la base de datos utilizada. Para usar JDBC, es necesaria la implementación específica de la base de datos del controlador JDBC.

La API de JDBC soporta la comunicación entre la aplicación Java y el *driver* que realizará la conexión entre la base de datos, es decir, actúa como intermediaria. JDBC es la API común con la que interactúa el código de nuestra aplicación. Debajo está el controlador compatible con JDBC para la base de datos que vamos a utilizar.

Para realizar la conexión, necesitaremos el *driver* JDBC, que es el que nos permitirá que nuestra aplicación interactúe con la base de datos. Según qué base de datos queramos utilizar, tendremos que usar el *driver* correspondiente.

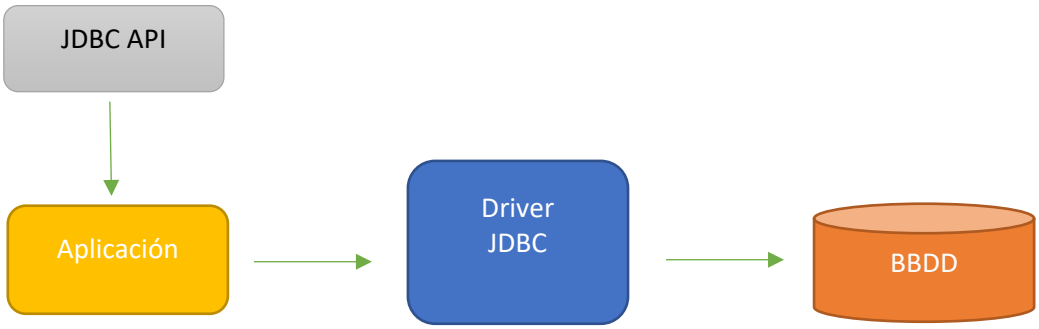


Ilustración 16. Esquema de conexión de base de datos en una aplicación Java.

Como ventajas, podemos destacar que es muy fácil de utilizar y permite conectarse a cualquier base de datos. Por otro lado, como desventajas podemos destacar que su rendimiento puede verse degradado y el *driver* se tiene que instalar en el cliente.

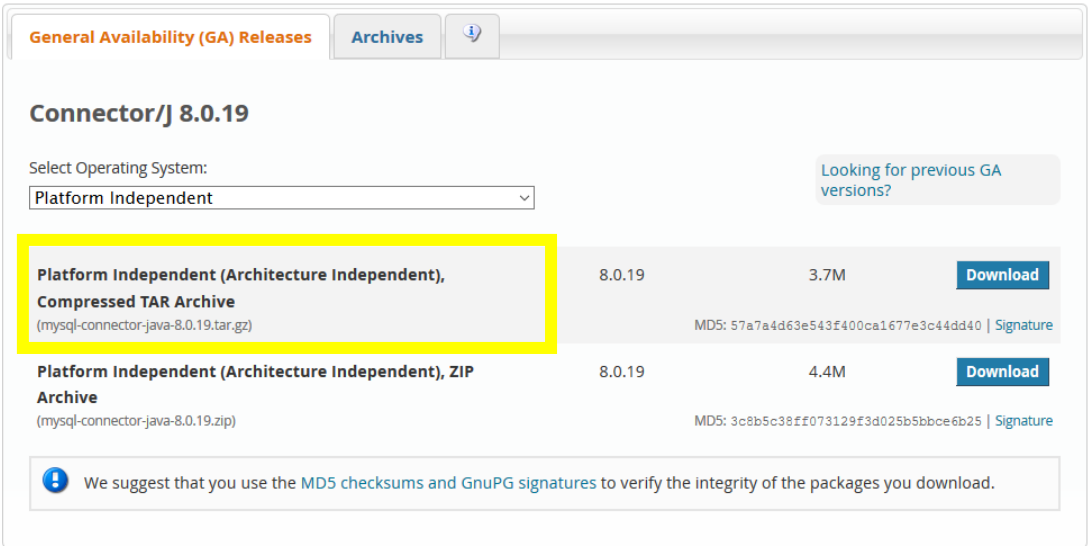


Ilustración 17. Indicación para descargar el jar del driver de MySQL.

2.3.2. Establecimiento de conexiones

Para entender mejor cómo se realiza la conexión, realizaremos un ejercicio explicativo.

> Recordad que todos los ejercicios los tenéis en GitLab, y os permitirá seguir mejor la explicación. Aquí tenéis el enlace:
>
> https://gitlab.com/ilerna/common/java

Para este ejercicio, utilizaremos una base de datos MySQL. El proceso de instalación de MySQL puede ser algo complicado. A continuación, detallaremos los pasos para tener correctamente instalada la base de datos y crear una nueva base de datos para nuestro proyecto. Tenemos que descargar el instalador de aquí:

> https://dev.mysql.com/downloads/installer/

Abriremos el instalador y se abrirá este *pop up* y seguiremos los pasos que vemos detallados en las capturas.

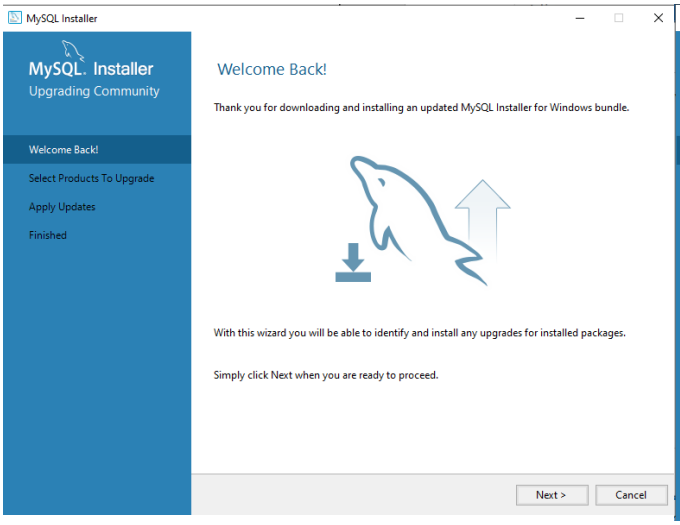


Ilustración 18. Paso 1: le damos a Next.

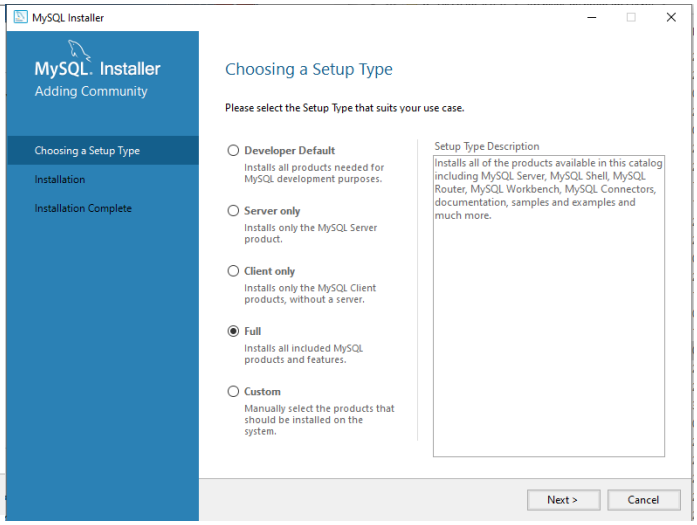


Ilustración 19. Paso 2: elegimos instalación Full y le damos a Next.

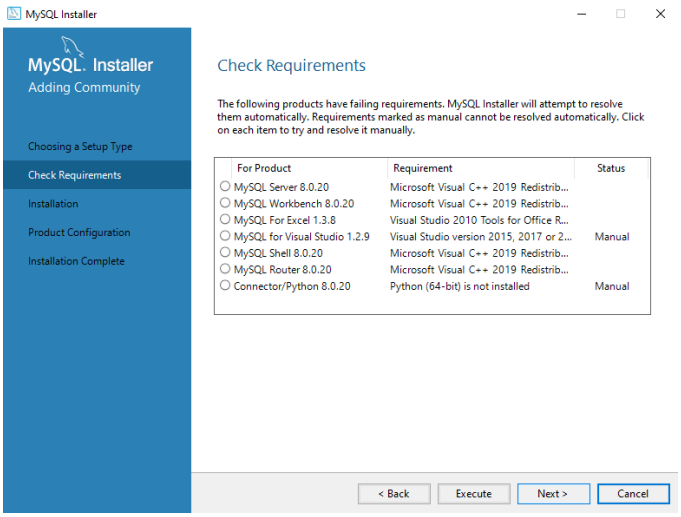


Ilustración 20. Paso 3: apretar botón Execute.

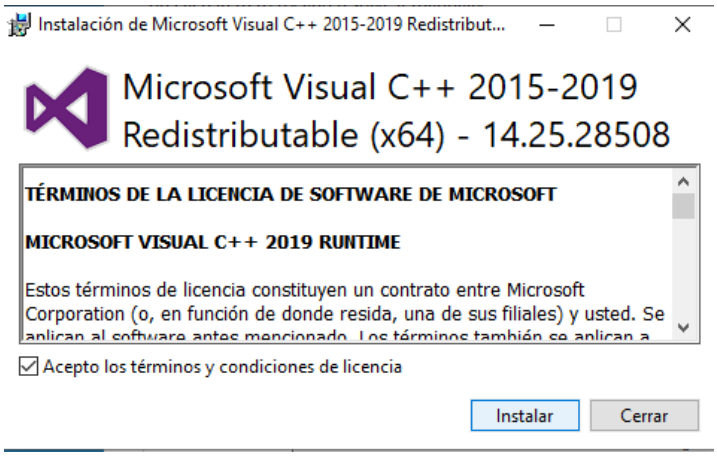


Ilustración 21. Paso 4: dar a Instalar si aparece el pop up.

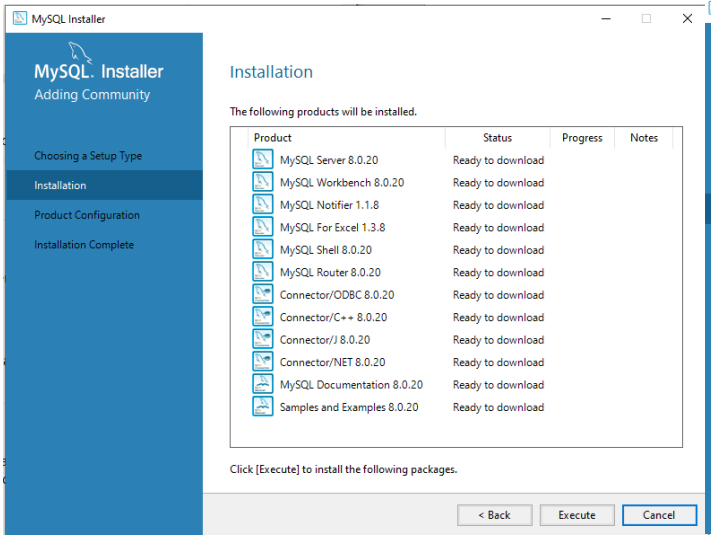


Ilustración 22. Paso 5: saldrá este pop up en el cual debemos seleccionar Execute.

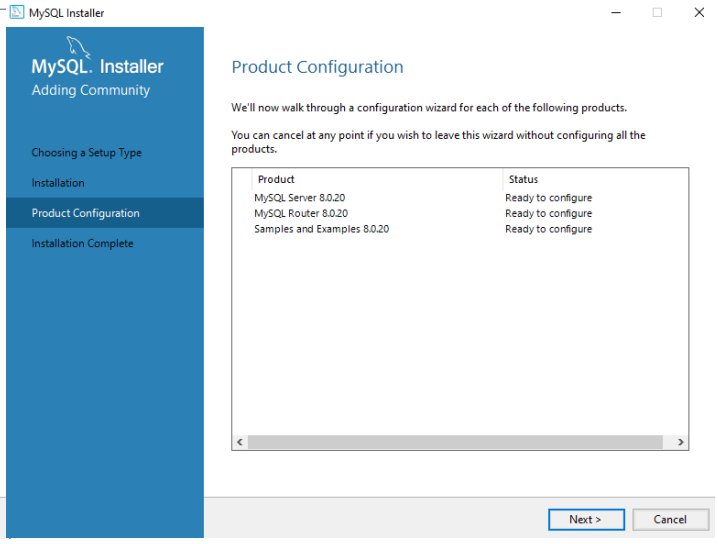


Ilustración 23. Paso 6: al acabar el paso anterior, saldrá este, solo hay que pulsar Next.

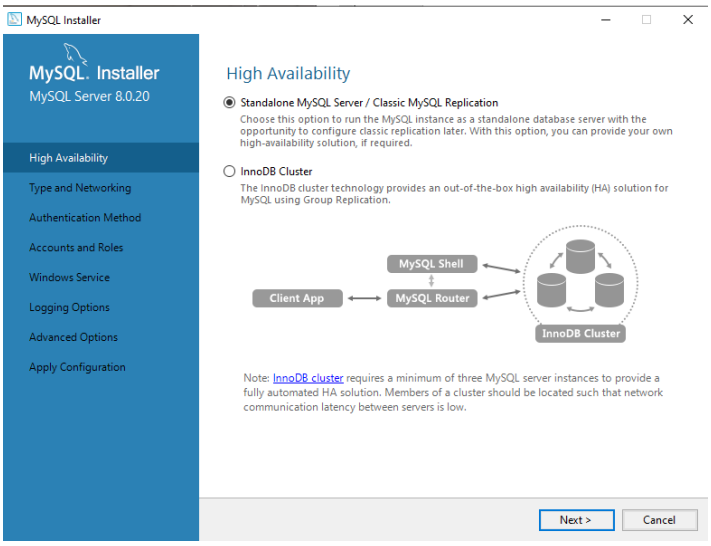


Ilustración 24. Paso 7: elegimos la selección que vemos por pantalla y damos a Next.

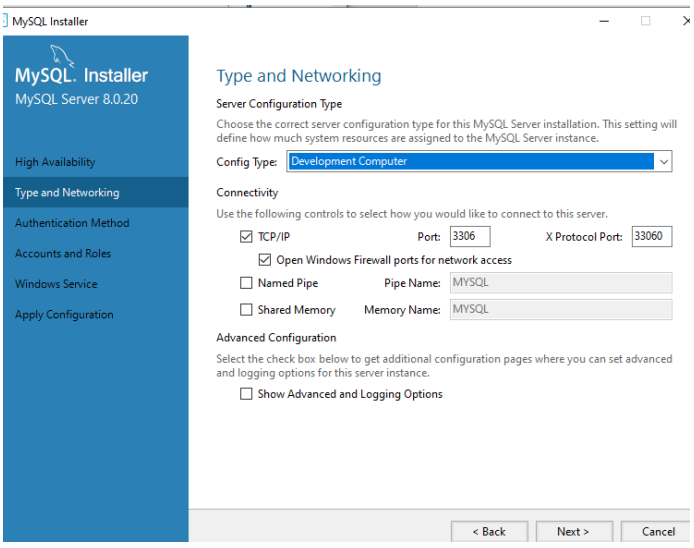


Ilustración 25. Paso 8: dejamos la configuración por defecto y damos a Next.

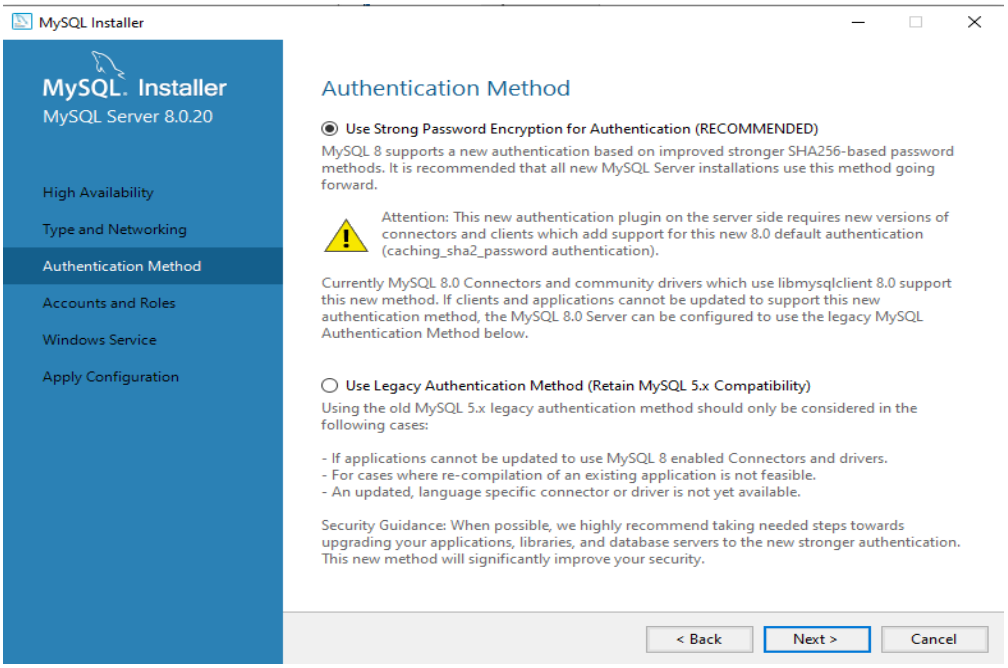


Ilustración 26. Paso 9: si nos sale esta pantalla, elegimos la opción de la pantalla y damos a Next.

Es importante prestarle atención a esta captura de pantalla, porque es la que nos permite establecer la contraseña de la base de datos. El usuario por defecto era "root", pero podemos elegir la contraseña que deseemos. Es importante guardar la contraseña porque la utilizaremos en el futuro para establecer la conexión a la base de datos.

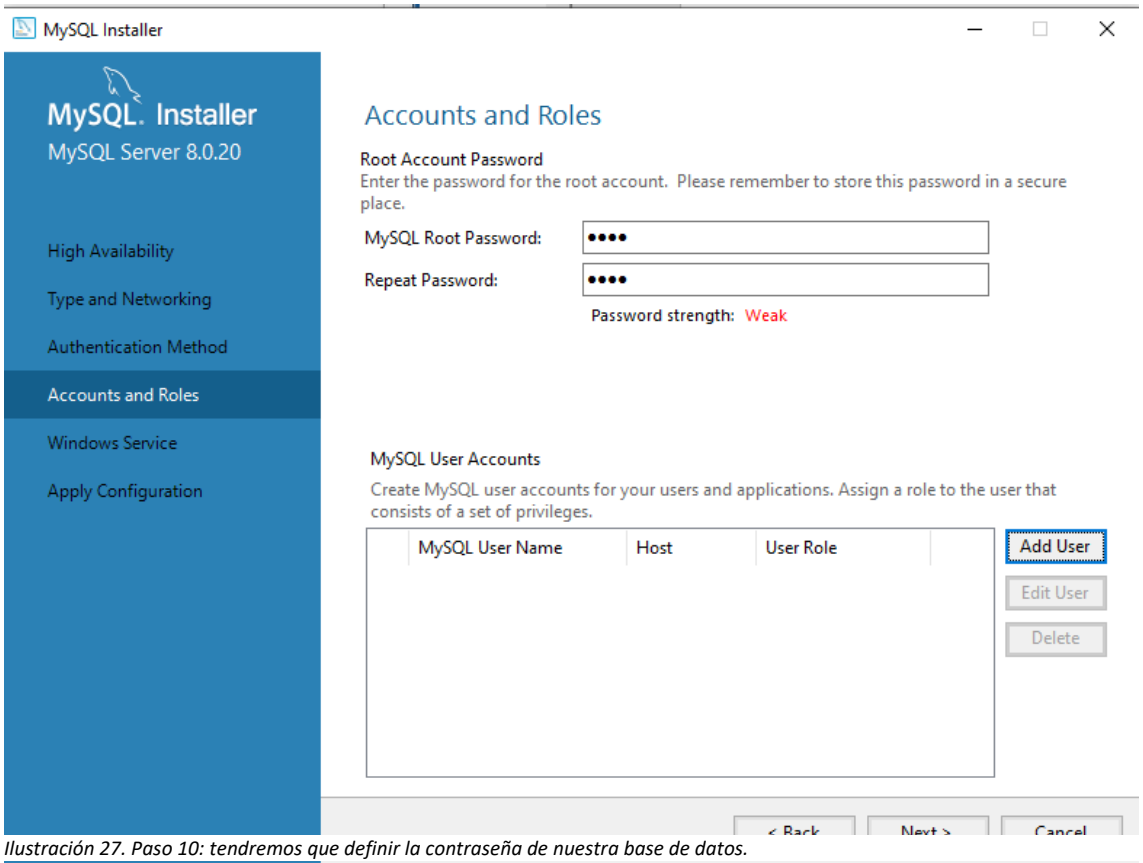


Ilustración 27. Paso 10: tendremos que definir la contraseña de nuestra base de datos.

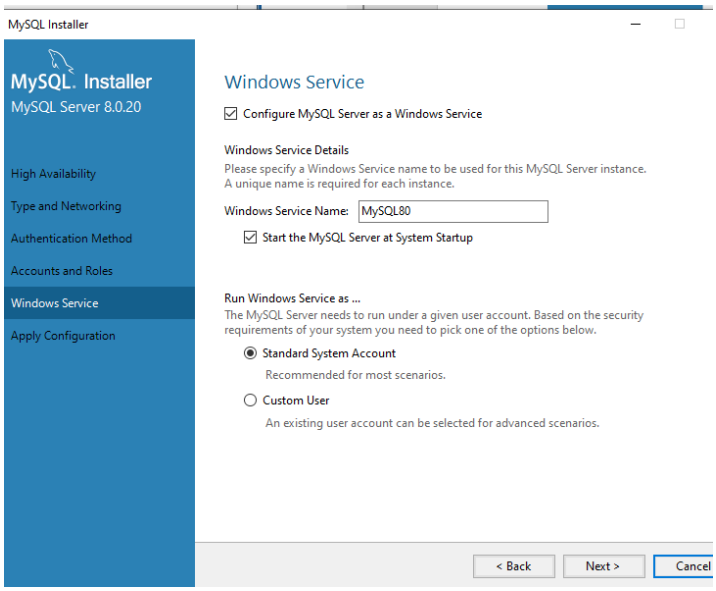


Ilustración 28. Paso 11: en esta pantalla dejamos valores por defecto o como en la captura y damos a Next.

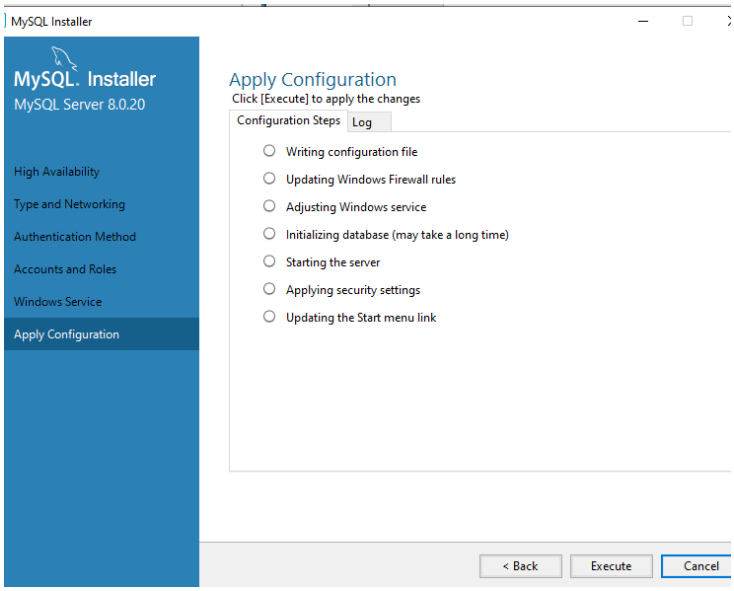


Ilustración 29. Paso 12: pulsamos el botón Execute.

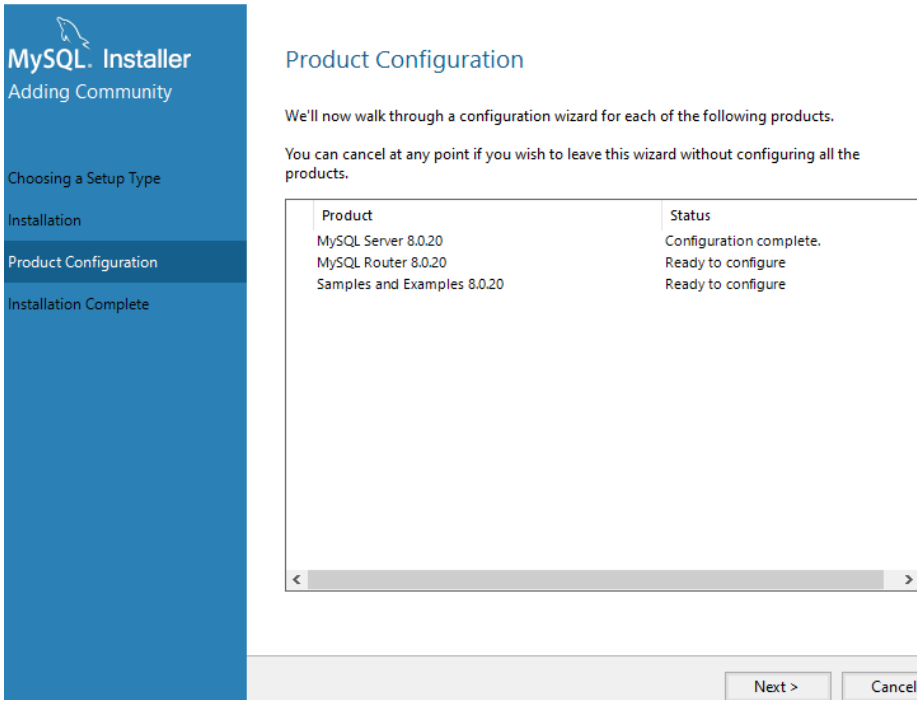


Ilustración 30. Paso 13: en esta pantalla solo tendremos que darle a Next.

A continuación, nos aparecerá esta pantalla. Servirá para comprobar la conexión. Tenemos que meter el usuario "root" y la contraseña que hayamos elegido unos pasos anteriores. Le damos al botón de *Check* y tiene que salir el mensaje "Connection succeded" como en la captura. Si no es así, es que no hemos introducido bien el usuario o la contraseña. Una vez que todo esté correcto, pulsamos sobre *Next*.

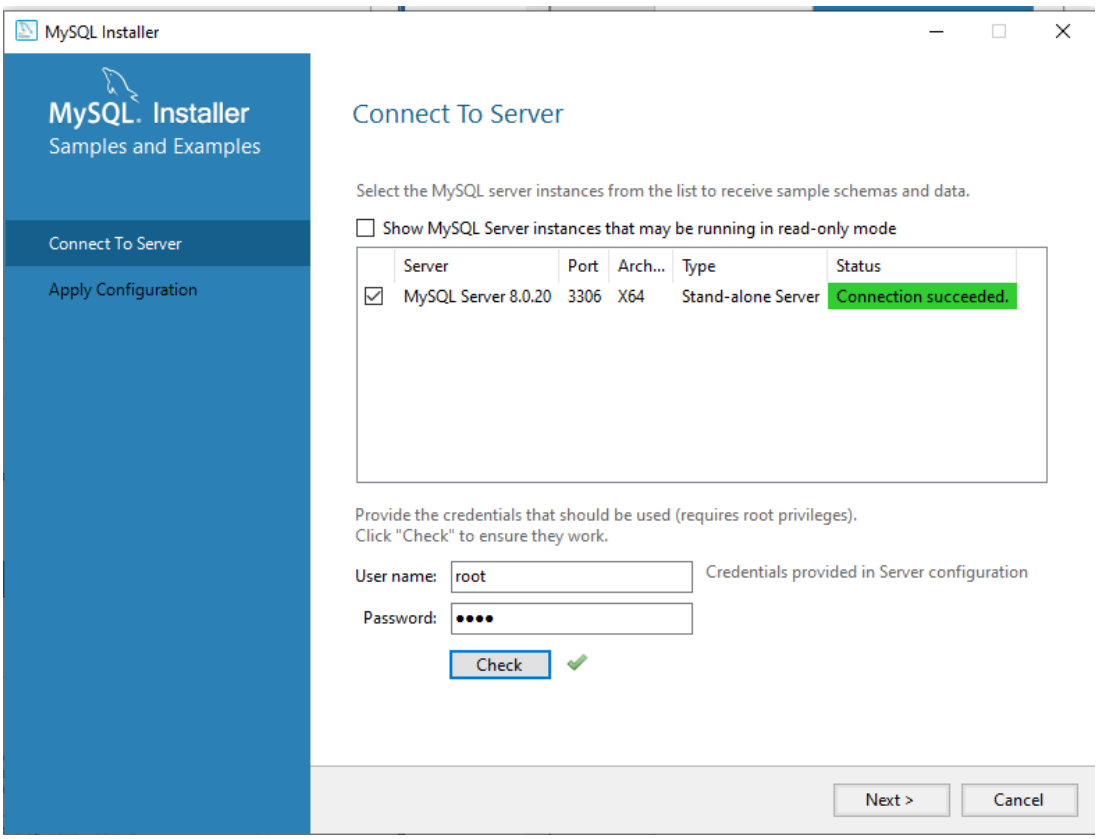


Ilustración 31. Paso 14: esta pantalla permite revisar la conexión con la BBDD.

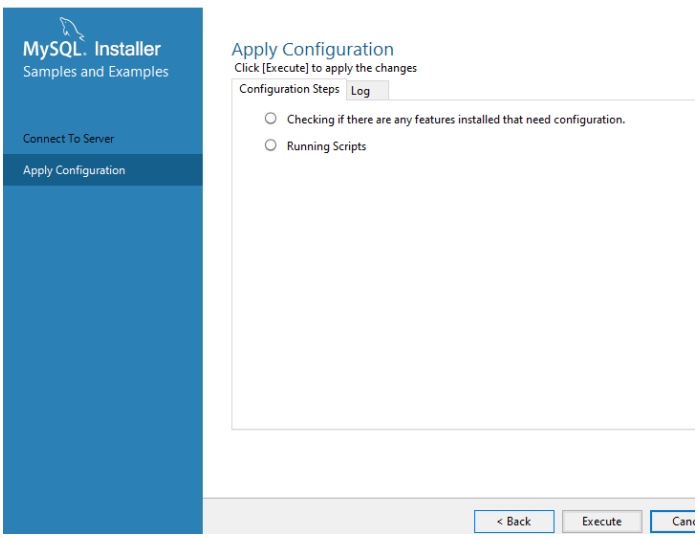


Ilustración 32. Paso 15: en esta pantalla le debemos pulsar Execute.

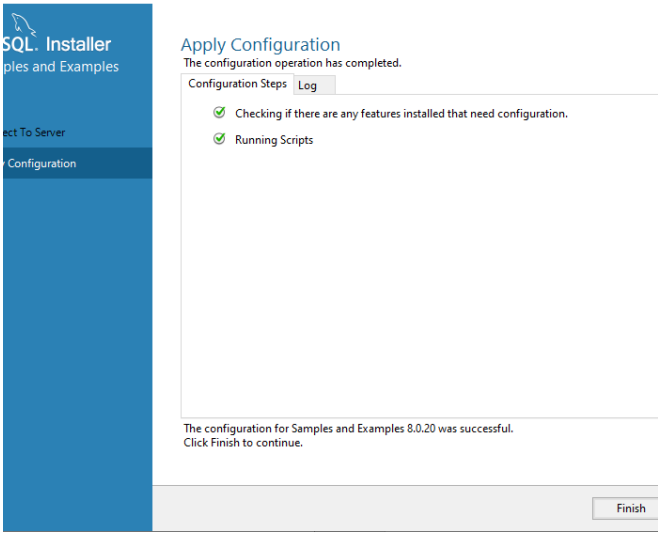


Ilustración 33. Paso 16: si todo va bien en el paso anterior, saldrá esta ventana y le damos a Finish

Después de todo esto, saldrá otra ventana en la cual solo debemos darle a *Next* y ya habremos acabado la instalación.

Una vez finalizada, deberemos crear la base de datos que utilizará nuestra aplicación. Cada aplicación debería tener su base de datos por separado para diferenciarla del resto y para mantener un orden.

Este paso lo debemos realizar mediante un *script*. Un ***script*** es un conjunto de sentencias SQL que ejecutan tareas en la base de datos para realizar ciertas acciones. En este caso, tendremos que crear la base de datos. Para ejecutar estas sentencias, deberemos abrir MySQL Client, que se ha instalado en nuestro ordenador durante la instalación de MySQL.

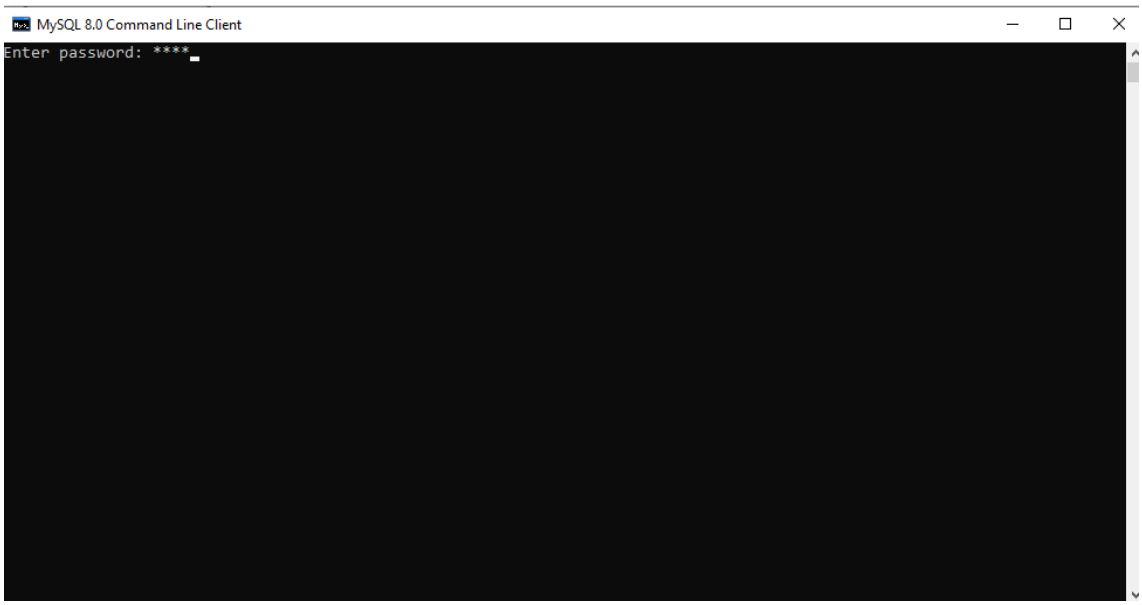


Ilustración 34. Captura de MySql Comand Line Client.

Pondremos la contraseña que hemos definido durante la instalación y ya podremos realizar la creación de nuestra base de datos. Los comandos que tendremos que ejecutar son:

```
CREATE DATABASE ilerna;
CREATE USER 'alumno'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON *.* TO 'alumno'@'localhost';
```

La primera línea crea la base de datos para nuestra aplicación. La segunda crea un usuario y le asigna una contraseña. El usuario que hemos creado al principio es el usuario administrativo de nuestra base de datos MySQL. Cuando creamos una base de datos para una aplicación, también debemos crear el usuario que gestionará esa base de datos. Por eso, creamos uno para la aplicación. A continuación, debemos darle permisos para realizar cualquier tipo de acción en la base de datos.

Después necesitaremos descargar el *driver* de MySQL que nos permitirá realizar la conexión con la base de datos y tendremos que añadirlo a nuestro proyecto. Lo descargaremos de este enlace:

https://dev.mysql.com/downloads/connector/j/

Después de añadir el *driver* a nuestro proyecto, es el momento de realizar la conexión usando JDBC. Podremos diferenciar tres pasos para realizar el proceso:

2.3.2.1. Registrar el *driver* JDBC

Cuando tengamos la clase creada de encargarse de las conexiones, tendremos que registrar el *driver* JDBC. Este proceso no es más que indicarle a la clase a qué *driver* tiene que apuntar. Al registrar el *driver*, se carga en memoria para poder ser utilizado por la interfaz JDBC.

```
try {
      Class.forName("com.mysql.jdbc.Driver");
}catch(ClassNotFoundException ex) {
      System.out.println("Error al cargar el driver.");
}
```

Como vemos en el ejemplo, para registrar el driver usaremos *Class.forName()*. Al llamar al método *forName()* lo que hacemos es cargar de manera dinámica el *driver* de la clase en memoria. Necesitamos pasar por parámetro el *driver* de la base de datos que vamos a usar. Esta llamada solo se va a realizar una vez.

Si no usamos una base de datos MySQL, el procedimiento es el mismo, pero le pasaremos el *driver* de la base de datos correspondiente.

2.3.2.2. Crear una URL de conexión a la base de datos

Después de cargar el *driver*, tendremos que establecer la conexión con la base de datos. Para ello, usaremos *DriverManager.getConnection()*. Al llamar a *getConnection()* tendremos que pasarle una URL indicando qué base de datos usaremos, el nombre, el usuario y la contraseña para que pueda acceder a la BBDD.

Según la BBDD que usemos, el formato de la URL será algo distinto. Aquí tenemos una muestra ejemplo de las más importantes:

RDBMS	Nombre *driver* JDBC	Formato URL
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName

Tabla 11. Tabla esquemática según BBDD para la formación de la URL de conexión.

Como podemos ver, la estructura para formar la URL es siempre parecida, pero tiene variaciones según la BBDD, por eso es importante mirar antes cómo formarla para establecer correctamente la conexión y así evitar posibles errores.

2.3.2.3. Crear los objetos de conexión

Preparar la conexión a la base de datos es un paso importante que tener en cuenta. Para realizar la conexión, es necesario utilizar la clase *Connection*. Este será el intermediario que conectará nuestra aplicación y la base de datos. Si estamos utilizando otro entorno

para realizar la conexión, se necesitará el nombre del *host* y el puerto. Para construir la URL, seguiremos esta estructura:

```
String url = "jdbc:mysql://hostname:port/ilerna";
```

```
String url = "jdbc:mysql://localhost/ilerna";
String usuario= "alumno";
String contrasena = "password"
Connection conn = DriverManager.getConnection(url, usuario,
contrasena);
```

Para realizar la conexión se utiliza *DriverManager* y se llamará al método *getConnection()*, que devolverá un objeto *Connection* con todos los datos de la conexión realizada en BBDD.

Ahora que sabemos cómo realizar la conexión, veremos un ejemplo práctico con todo lo que hemos aprendido hasta ahora.

```
Connection connection;
String url = "jdbc:mysql:tema2";
String usuario = "usuario";
String contrasena = "pasword";

try {
      Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection(url,usuario,
      contrasena);
           // Acceso a datos utilizando el objeto de conexión
} catch (SQLException sqle) {
       // Trataremos el error si no se establece conexión
} finally {
      try {
          connection.close();
      } catch (SQLException e) {
      // Trataremos el error si no podemos cerrar la conexión
      }
}
```

En la muestra de código anterior, podemos ver una simplificación de lo que sería una conexión sencilla. En primer lugar, se deberá declarar el objeto *Connection*, sin asignar valor. Este objeto es el encargado de recoger el resultado de la conexión que realiza la clase *DriverManager*. A continuación, se definirá la URL a la BBDD creada (en esta

ocasión es "tema2", pero en vuestro lugar será el nombre de vuestra base de datos), el usuario y la contraseña.

Una vez realizada la conexión, si se produce un error se debe recoger en un *SQLException*, tal y como mostramos en el ejemplo, y recordad de cerrar la conexión: debemos cerrar los recursos de la aplicación, pues dejarla abierta nos dará problemas en futuras consultas.

Para poner en práctica los conceptos explicados, vamos a crear nuestra base de datos.

```
public class ConnectorBBDD {
//Declaramos los diferentes objetos que nos permitirán realizar
la conexion
      private Connection connect = null;
      // JDBC driver
      static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
      static final String DB_URL = "jdbc:mysql://tema2";
      static final String USUARIO = "usuario";
      static final String CONTA = "password";

      public Connection conector() throws Exception {
            try {
            // Esto se encarga de cargar el driver de MySQL
            Class.forName(JDBC_DRIVER);
            // Establecemos la conexion
            Connection connect =
            DriverManager.getConnection(DB_URL + "user=" +
            USUARIO + " &password=" + CONTA);

            } catch (Exception e) {
                  throw e;
            } finally {
                  connect.close();
            }
            return connect;
      }
}
```

Vamos a crear una clase auxiliar que llamaremos "ConnectorBBDD". El propósito de esta clase será realizar todos los pasos necesarios para establecer la conexión con la base de datos que vayamos a utilizar.

Como vemos, hemos creado un método *connector()* que devuelve un objeto *Connection*. Este método se encargará de registrar el *driver* y seguidamente declararemos una nueva instancia de la clase *Connection*. Para realizar la conexión, deberemos llamar al método *DriverManager.getConnection()* pasándole la URL de la base de datos, el usuario y la contraseña. Estas variables las tenemos que declarar fuera del método con la

información de nuestra base de datos. Si todo va bien, devolverá un objeto *Connection* con los datos de la conexión. Para poder establecer la conexión con *getConnection()*, existen otras formas utilizando el mismo método pero con diferentes parámetros:

Método	Descripción
getConnection(String url)	Establecerá la conexión con tan solo la URL. La URL deberá contener toda la información necesaria para conectarse a la base de datos.
getConnection(String url, Properties info)	Este método se encargará de realizar la conexión a partir de la URL y un *Properties*. Esta clase contendrá toda la información necesaria para conectarse a la base de datos, como el usuario o la contraseña.

Debemos tener en cuenta que cabe la posibilidad de que se produzca un error en este proceso, por este motivo hemos englobado todos los pasos dentro de un *try-catch* que controlará el error si se produce. Un posible error podría ser un error de conexión debido a que hemos definido incorrectamente la conexión, a que el usuario o la contraseña están mal o a que no hemos realizado el proceso de registrar el *driver*. En todo caso, si se produce un error, entrará en el bloque *catch*.
Para finalizar, vemos que tenemos un bloque *finally* para cerrar la conexión del objeto.

2.3.3. Definición de objetos destinados al almacenamiento del resultado de operaciones con bases de datos. Eliminación de objetos una vez finalizada su función

Ya realizada la conexión, nuestra aplicación está preparada para interactuar con la base de datos. En este apartado os explicaremos qué objetos serán los más adecuados para almacenar la información de nuestras operaciones con la base de datos. Podemos destacar estos tres como los más importantes:

Nombre de la interfaz	Descripción
PreparedStatement	Se utiliza para las operaciones SQL con parámetros dinámicos. Esta interfaz permite pasar parámetros a sus métodos y ejecutar las operaciones más de una vez.

Statement	Se utiliza de manera general para cualquier tipo de operación con la base de datos. Es especialmente útil con SQL sin valores dinámicos.
ResultSet	Esta interfaz se caracteriza por devolver la información en forma de tabla.
CallableStatement	Se utiliza para trabajar con procedimientos.

Tabla 12. Interfaces más importantes para las operaciones con bases de datos.

2.3.3.1. Interfaz *Statement*

Este tipo de interfaz se utiliza para conexiones de carácter general. Es bastante útil cuando queremos usar consultas estáticas SQL.

> Una **consulta estática** es aquella operación SQL que no varía la consulta en ningún momento, siempre será la misma y los valores de la consulta no cambiarán.

Este tipo de clase no acepta parámetros. Las sentencias más utilizadas son todas aquellas que no necesitan datos de los objetos Java, como, por ejemplo, cualquier *create*, *insert*, *delete*, *alter* o *drop*.

La implementación sería algo más o menos así:

```
Statement stmt = null;
private Connection connect = null;
// JDBC driver name and database URL
private String JDBC_DRIVER = "com.mysql.jdbc.Driver";
private final String DB_URL = "jdbc:mysql://localhost/ilerna";
private String USUARIO = "alumno";
private String CONTA = "password";
try {
      Class.forName(JDBC_DRIVER);
      // Establecemos la conexion
      connect = DriverManager.getConnection(DB_URL + "user=" + USUARIO
+ " &password=" + CONTA);
      System.out.println("Nos hemos conectado a la BBDD");
      stmt = connect.createStatement();
      String sql = "ALTER DATABASE ilerna MODIFY NAME = tema2";
      stmt.executeUpdate(sql);
}
catch (SQLException e) {
//continuación codigo
}
finally {
   stmt.close();
}
```

Una vez creado el objeto, podemos usarlo para ejecutar consultas SQL. Este objeto tiene tres métodos importantes para ello:

- *execute():* se usará para ejecutar consultas dinámicas SQL.
- *executeUpdate():* se usa cuando hacemos *insert*, *delete* o *update*.
- *executeQuery():* se usa para realizar *select*.

Debemos tener en cuenta que, cada vez que creemos este tipo de objeto, tendremos que cerrarlos, por lo que es necesario utilizar el método *close()* para cerrar la conexión.

2.3.3.2. Creación de objetos *PreparedStatement*

Esta interfaz hereda de *Statement*, y nos dará mucha más funcionalidad respecto del objeto anterior. Este objeto nos ofrecerá más flexibilidad, permitiendo pasarle

dinámicamente argumentos al objeto. Podremos usar todas las sentencias mencionadas anteriormente, igual que en el objeto *Statement*.

```
PreparedStatement pstmt = null;
try {
   Estudiante estudiante = new Estudiante();
   String SQL = "select from Estudiante where id=? And apellido
= ?";
   pstmt = conn.prepareStatement(SQL);
   pstmt.setInt(1, estudiante.getId());
   pstmt.setString(2, estudiante.getApellido());
//continuación de código
}catch (SQLException e) {
   //continuación de código
}
finally {
     pstmt.close();
`
```

En JDBC, los parámetros que vemos en la sentencia SQL están representados por un símbolo de interrogación, conocido como **marcador de parámetro**. Tendremos que proporcionar valores para cada parámetro antes de ejecutar la consulta. Estos valores los podremos asignar con los datos de nuestro objeto Java, en este caso, el objeto "Estudiante" con los *getters getId()* y *getApellido()*, que recogerán el valor de este atributo y nos servirán para asignárselo de manera dinámica. Si no se asignan los argumentos, se lanzará una excepción de tipo *SQLException*.

Este tipo de objeto también dispone de los mismos métodos que el objeto que hemos visto en el apartado anterior, pero, como diferencia, a estos métodos podemos pasarles parámetros.

- *execute(String sql):* se usa para ejecutar consultas dinámicas SQL.
- *executeUpdate(String sql):* se usa cuando hacemos *insert*, *delete* o *update*.
- *executeQuery(String sql):* se usa para realizar *select*.

Si nos fijamos, al final del ejemplo también tenemos un *close()* para este tipo de objetos, ya que no es diferente de lo mencionado anteriormente. Siempre que abramos un objeto para interactuar con una BBDD, tendremos que cerrarlo al final de su uso.

2.3.3.3. Creación de objetos *ResultSet*

ResultSet es otra interfaz que podemos utilizar para recoger el resultado de una de nuestras consultas. Solo podemos utilizar este tipo de objeto para seleccionar datos, ya que no permite actualizar datos, es decir, solo se usa en las sentencias *select*. Se

caracteriza por devolver una tabla de datos según columnas y filas. Para obtener los datos de la consulta, se tendrá que acceder con un *get* más el tipo de dato al que queramos acceder. Por ejemplo, *getInt* devolverá un *integer*, mientras que un *getString* devolverá un *varchar*.

Aquí tenemos una relación de datos que podemos obtener en nuestras consultas.

Tipo de get	Tipo de dato de la BBDD
getInt	*Integer*
getLong	*Big int*
getFloat	*Real*
getDouble	*Float*
getBignum	*BIT*
getString	*Varchar/char*
getDate	*Date*
getTime	*Time*
getTimestamp	*Time stamp*
getObject	Cualquier otro tipo

Tabla 13. Tabla con los get más comunes del objeto ResultSet.

Se caracteriza también por solo avanzar hacia delante con los datos que obtiene, por tanto, no podremos recorrerlos hacia atrás. Para ver cómo funciona, aquí mostramos un breve ejemplo de implementación.

```
//Paso 1 establecer conexión + pasar la consulta
PreparedStatement s= con.prepareStatement("select id, dni, nombre
from Estudiantes");
//Paso 2. Definir resultSet y ejecutar la consulta
ResultSet resultado = s.executeQuery();
//Paso 3. Imprimir el resultado
while(resultado.next()){
      System.out.printIn("Id: " + resultado.getInt());
      System.out.printIn("dni: " + resultado.getString());
      System.out.printIn("nombre: " + resultado.getString());
}
//Paso 4: Cerramos los objetos que usamos para realizar la
conexión y obtener resultado
resultado.close();
```

Tal y como apreciamos en el ejemplo, primero necesitamos un objeto *PreparedStatement* para realizar la conexión y preparar la consulta. A continuación, creamos un *ResultSet* que nos permitirá ejecutar la consulta y que es donde guardaremos los datos obtenidos. Como vemos, para ejecutar la consulta solo necesitamos que el objeto *PreparedStatement* llame al método *executeQuery()* y que el objeto *ResultSet* recoja el resultado. Con el método *next()* recorremos los resultados hacia delante y podemos ir accediendo a ellos según su tipo, como hemos explicado anteriormente.

2.3.4. Ejecución de sentencias de descripción de datos

Entendemos como **sentencias de descripción de datos (*data definition language)*** todas aquellas que permitan manipular la estructura de nuestra base de datos.

Se trata de una manera de definir ciertos comandos de datos. Estas sentencias pueden ser:

- *CREATE:* se encarga de crear bases de datos o tablas.
- *DROP:* la usaremos para borrar bases de datos o tablas.
- *ALTER:* la usaremos para la modificación de tablas o bases de datos.
- *RENAME:* usaremos esta sentencia para cambiar el nombre de nuestra base de datos o de una tabla.

Estos comandos son exclusivamente para hacer modificaciones en las tablas, no para realizar consultas en MySQL. JDBC nos da opciones para poder hacer este tipo de sentencias mediante el código Java.

2.3.4.1. Crear tablas o bases de datos

El comando *CREATE* se puede usar para crear una tabla o una base de datos. Normalmente, ya tendremos creada la base de datos, pero si en alguna ocasión se tiene que realizar, se puede hacer mediante la aplicación. Aquí tenemos la estructura teórica que podemos adaptar para nuestras sentencias y necesidades. La estructura básica de estas sentencias son estas, a modo de recordatorio:

```
//Para crear una base de datos
CREATE DATABASE nombre_base_datos;
//Para crear una tabla
CREATE TABLE nombre_tabla (
    Columna_1 tipo_dato,
    Columna_2 tipo_dato,
    ...
);
```

Si tenéis dudas de los tipos de datos permitidos en la creación de sentencias SQL, aquí podéis encontrar un enlace con el resumen de los que existen:

https://www.w3schools.com/sql/sql_datatypes.asp

Con *CREATE DATABASE* podremos crear nuevas bases de datos. Dentro de esa base de datos, podremos crear las tablas para nuestra aplicación. Se crearán bases de datos nuevas siempre que sea necesario, aunque lo más habitual es crear nuevas tablas.

En primer lugar, hemos creado un método que se encargará de realizar la creación de una nueva base de datos. Seguidamente, se define *Connection*, que nos permitirá realizar la conexión a la base de datos con nuestra clase auxiliar. Este es el paso 1 que vemos en el ejemplo.

En segundo lugar, tenemos un objeto *Statement*. Como hemos explicado en apartados anteriores, este tipo es muy útil para cualquier tipo de operación con base de datos y es el más simple que existe, por este motivo elegimos definir esta clase. Cuando la conexión esté realizada con el objeto *Connection*, podremos llamar al método *createStatement()*, que preparará un objeto *Statement* para poder enviar nuestras operaciones SQL a la base de datos.

A continuación, tenemos que crear en un *string* la sentencia SQL que permitirá crear la base de datos. Y una vez creado, le pasaremos el *string* al método *executeUpdate()* del objeto *Statement* creado.

```
public void crecionBaseDeDatos() throws Exception {
	Connection conn = null;
	Statement stmt = null;
	try {
		//Paso 1.Previamente habremos realizado la conexión
		conn = conector.conector();
		//Paso 2. Creamos un nuevo objeto con la conexión
		stmt = conn.createStatement();
		//Paso 3. Definimos la sentencia de crear una nueva
		base de datos
		String sql = "CREATE DATABASE ejemplo";
		//Paso 4. Ejecutar la sentencia
		stmt.executeUpdate(sql);
	}catch(SQLException se){
		//Gestionamos los posibles errores que puedan surgir
		durante la ejecucion de la insercion
		se.printStackTrace();
	}catch(Exception e){
		//Gestionamos los posibles errores
		e.printStackTrace();
		}finally{
			//Paso 5. Cerrar el objeto en uso y la conexión
			stmt.close();
			conn.close();
		}
	}
}
```

Para la creación de tablas, el proceso es exactamente el mismo, pero cambia el tipo de sentencia utilizado.

En todos los casos, debemos tener en cuenta que tenemos que englobar todo el procedimiento en un *try-catch* para controlar los posibles errores que se puedan producir. Los más usuales serán:

- Error de conexión a la base de datos.
- Error en la sentencia SQL que hemos definido.
- Error en cerrar los objetos en uso.

Para ello, debemos utilizar *SQLException* como el primer *catch*, ya que este tipo de error es más específico. En segundo lugar, deberíamos tener un *Exception* para poder controlar cualquier otro error que no tenga que ver con la base de datos.

```
public void crecionTabla() throws Exception {
	Connection conn = null;
	Statement stmt = null;
	try {
		conn = conector.conector();
		stmt = conn.createStatement();
		String sql ="CREATE TABLE estudiante (" +
			"  id INT PRIMARY KEY AUTO_INCREMENT;" +
			"  dni VARCHAR(50) NOT NULL," +
			"  nombre VARCHAR(50) NOT NULL," +
			"  apellido VARCHAR(100)," +
			"  edad INT DEFAULT 10;" +
			");";
		stmt.executeUpdate(sql);
	}catch(SQLException se){
		se.printStackTrace();
	}catch(Exception e){
		e.printStackTrace();
		}finally{
			stmt.close();
			conn.close();
		}
	}
}
```

2.3.4.2. Modificar tablas o bases de datos

Para la modificación de tablas o de bases de datos utilizaremos la sentencia *ALTER*. *ALTER* es la operación SQL que se encargará de realizar cambios dentro de la base de datos o de la tabla que especifiquemos, siempre que esos datos existan. Nos permitirá añadir, cambiar o eliminar campos de una tabla, o también renombrar una tabla. Estas son las estructuras básicas para realizar este tipo de operaciones.

```
//Para modificar la base de datos
ALTER DATABASE nombre_base_de_datos MODIFY NAME=nuevo_nombre;
```

```
//Para añadir columnas
ALTER TABLE nombre_tabla ADD COLUMN nombre_columna tipo_dato;
//Para modificar columnas
ALTER TABLE nombre_tabla MODIFY COLUMN nombre_columna tipo_dato;
//Para borrar una columna
ALTER TABLE nombre_tabla DROP COLUMN nombre_columna;
```

Las opciones con esta sentencia son bastante amplias: podremos modificar columnas, añadirlas o hasta borrarlas. Aquí podemos ver un par de ejemplos prácticos:

```
public void modificarBaseDatos() throws Exception {
	Statement stmt = null;
	try {
		conn = conector.conector();
		System.out.println("Nos hemos conectado a la BBDD");
		stmt = conn.createStatement();
		String sql = "ALTER DATABASE ilerna MODIFY NAME =
		tema2";
		stmt.executeUpdate(sql);
	}catch (Exception e) {
		System.out.println("Se ha producido un error.");
	}
	finally {
		stmt.close();
		conn.close();
	}
}
```

Como vemos, en este ejemplo seguimos la misma estructura que en el apartado anterior, solo modificamos la SQL. Esta modificará el nombre de nuestra base de datos a "tema2".

```
Statement stmt = null;
try {
	//Paso 1: Realizamos la conexión
	conn = conector.conector();
	System.out.println("Nos hemos conectado a la BBDD");

	//Paso 2: Preparamos el objeto Statement
	stmt = conn.createStatement();
	//Paso 3:Modificacion de la base de datos, borrar 2
	columnas
	String sql = "ALTER TABLE estudiante DROP COLUMN dni, DROP
	COLUMN edad;";
	stmt.executeUpdate(sql);
}catch(SQLException se){
	//Gestionamos los posibles errores que puedan surgir
	durante la ejecución de la inserción
	se.printStackTrace();
} catch (Exception e) {
	System.out.println("Se ha producido un error.");
}
finally {
	stmt.close();
	conn.close();
}
```

En primer lugar, tenemos una sentencia que se encargará de borrar dos columnas. Se puede especificar borrar dos columnas una después de la otra, con una separación de una coma. Para modificar una tabla el procedimiento es el mismo: utilizamos siempre el objeto *Statement* y ejecutamos el método *executeUpdate()*.

Podemos sustituir la sentencia SQL por cualquier otra, como podrían ser una de estas dos:

```
sql = "ALTER TABLE estudiante ADD Email varchar(255);";
sql = "ALTER TABLE estudiante COMMENT = 'Comentario de prueba'";
```

En la primera solo ponemos una descripción a la tabla "estudiantes", y en la segunda ejecutamos la sentencia *ALTER TABLE* para añadir una nueva columna. Solo son ejemplos de posibles usos de la sentencia *ALTER TABLE*.

Como veis, es bastante fácil y versátil y admite muchas otras opciones. Las posibilidades son bastante amplias, según lo que necesitemos. La estructura en nuestra clase Java debe ser la misma que hemos ido planteando a lo largo de los temas.

2.3.4.3. Modificar tablas o bases de datos

El comando *DROP* es utilizado para borrar bases de datos o tablas dentro de nuestra base de datos. Cuando ejecutamos este comando, todos los datos que contenga la BBDD o la tabla también se borran. Como recordatorio, aquí tenemos la estructura básica para las dos opciones de *DROP:*

```
//Para borrar una base de datos
DROP DATABASE nombre_base_datos;
//Para borrar una tabla
DROP TABLE nombre_tabla;
```

La estructura es muy simple y adaptarlo a la estructura JDBC será parecido a lo que venimos realizando con los otros comandos. Como podemos apreciar, también usaremos la interfaz *Statement*. La sentencia *DROP*, al no devolver ningún registro ni información, es ideal para este tipo de interfaz, ya que está especialmente diseñada para operaciones sencillas con la base de datos.

```
Statement stmt = null;
try {
	// Paso 1: Realizamos la conexión
	conn = conector.conector();
	System.out.println("Nos hemos conectado a la BBDD");
	// Paso 2. Crear objeto y llamar a la conexión
	stmt = conn.createStatement();
	// Paso 3. Crear estructura de la sentencia
	String sql = "DROP TABLE estudiante";
	// Paso 4. Ejecucion
	stmt.executeUpdate(sql);
} catch (SQLException e) {
	e.printStackTrace();
} catch (Exception e) {
	e.printStackTrace();
} finally {
	// Paso 5. Cerrar objetos abiertos
	try {
		stmt.close();
	} catch (SQLException e) {
		e.printStackTrace();
	}
}
```

Para borrar del todo una base de datos, seguiremos el mismo procedimiento. Todas las tablas de esa base de datos se borrarán del todo. No es una sentencia que vayamos a usar a menudo, pero está bien conocerla.

```
Statement stmt = null;
try {
	// Paso 1: Realizamos la conexión
	conn = conector.conector();
	System.out.println("Nos hemos conectado a la BBDD");
	// Paso 2. Crear objeto y llamar a la conexión
	stmt = conn.createStatement();
	// Paso 3. Crear estructura de la sentencia
	String sql = "DROP DATABASE ilerna;";
	// Paso 4. Ejecucion
	stmt.executeUpdate(sql);
} catch (SQLException e) {
	e.printStackTrace();
} catch (Exception e) {
	e.printStackTrace();
} finally {
	// Paso 5. Cerrar objetos abiertos
	try {
		stmt.close();
	} catch (SQLException e) {
		e.printStackTrace();
	}
}
```

2.3.4.4. Renombrar tablas

El comando *RENAME* nos permitirá renombrar una tabla, es decir, modificar el nombre por otro. Este tipo de operación no producirá ningún cambio en los registros que contenga esa tabla, solo se modificará el nombre de la tabla. La estructura que deberemos seguir es esta:

```
//Para renombrar una tabla
RENAME TABLE nombre_tabla TO Nuevo_nombre_tabla;
```

Como hemos visto en ejemplos anteriores, el procedimiento es el mismo, solo cambia la sentencia SQL.

```
Connection conn = null;
Statement stmt = null;
try {
      // Paso 1: Realizamos la conexion
      conn = conector.conector();

      //Paso 2. Crear objeto y llamar a la conexión
      stmt = conn.createStatement();
      //Paso 3. Crear estructura de la sentencia
      String sql = "RENAME TABLE ilerna to prueba;";
      //Paso 4. Ejecución
      stmt.executeUpdate(sql);

}catch(SQLException se){
                  se.printStackTrace();
}catch(Exception e){
                  //Gestionamos los posibles errores
                  e.printStackTrace();
}finally{
      //Cerramos la connexion
      try{
            if(stmt!=null)
                  stmt.close();
                  conn.close();
      }catch(SQLException se){
            System.out.println("No se ha podido cerrar la
conexión.");
      }
}
```

2.3.5. Ejecución de sentencias de modificación de datos

Entendemos como sentencias de **modificación de datos** todas aquellas que actualicen o modifiquen de algún modo uno o más registros en una tabla. Este tipo de comando

SQL contempla la manipulación de los datos presentes en la base de datos, y serán prácticamente la gran mayoría de los comandos conocidos por SQL:

- *INSERT:* añadirá una fila en la tabla de base de datos indicada.
- *DELETE:* se encarga de borrar información de una tabla de la base de datos.
- *UPDATE:* se encarga de modificar los registros de una tabla de la base de datos.

Estas sentencias nos permitirán borrar datos, insertar nuevos registros o actualizar uno existente. El procedimiento para crear estas sentencias dentro de nuestra aplicación Java es parecido a los ejemplos anteriores, cambiará la consulta. Vamos a ver algunos ejemplos:

2.3.5.1. Inserción de datos en una tabla

Esta es una sentencia que nos permitirá insertar nuevos registros en una tabla de nuestra base de datos. Esta será una de las opciones más usadas. La implementación en nuestra aplicación será de forma muy parecida a los ejemplos ya explicados con anterioridad.

```
//Para insertar datos a la bbdd
INSERT INTO nombre_tabla (nombre_columna, nombre_columna) VALUES
(valor, valor);
```

Como vemos, para realizar un nuevo registro también tenemos que crear un objeto *Statement* que nos servirá para llamar al método *createStatement()*. La sintaxis del *INSERT* la crearemos mediante un *string*, tal y como veníamos haciendo hasta ahora, y se lo pasaremos al método *executeUpdate()*.

```
Connection conn = null;
PreparedStatement stmt = null;
try {
      //Utiliza la clase auxiliar que hemos creado para
      establecer conexión con bbdd
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");

      String sql = "INSERT INTO Estudiante (id, dni, nombre,
      apellido, edad) VALUES (22, '11111111H', 'Zara', 'Ali',
      18)";
      stmt.executeUpdate(sql);
}catch(SQLException se){
      se.printStackTrace();
}catch(Exception e){
      e.printStackTrace();
}finally{
      try{
            if(stmt!=null)
                  stmt.close();
                  conn.close();
      }catch(SQLException se){
            System.out.println("No se ha podido cerrar la
            conexión.");
      }
}
```

Existe otra posibilidad de inserción de datos, con datos dinámicos de nuestra aplicación. Las aplicaciones suelen interactuar con los usuarios que las utilizan, por tanto, muchos de los datos serán cambiantes y esa información nunca será del todo fija.

Por ello, existe la posibilidad de realizar sentencias SQL dinámicas. La estructura será muy parecida a lo que estábamos usando hasta ahora, pero en lugar del objeto *Statement* usaremos el objeto *PreparedStatement*. Este tipo de objeto está especialmente diseñado para poder realizar operaciones con sentencias dinámicas.

```
Connection conn = null;
PreparedStatement stmt = null;
try {
      //Utiliza la clase auxiliar que hemos creado para
      establecer conexión con bbdd
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");

      String sql = "INSERT INTO Estudiantes (id, dni, nombre,
      apellido, edad) VALUES (?, ?, ?, ?, ?)";
      //Imaginemos que viene con datos
      Estudiante estudiante = new Estudiante();

      //Prepararemos la query para que coja los datos de manera
      dinamica.
      stmt = conn.prepareStatement(sql);
      stmt.setInt(1, estudiante.getId());
      stmt.setString(2, estudiante.getDni());
      stmt.setString(3, estudiante.getNombre());
      stmt.setString(4, estudiante.getApellido());
      stmt.setInt(5, estudiante.getEdad());
      stmt.executeUpdate(sql);
}catch(SQLException se){
      se.printStackTrace();
}catch(Exception e){
      e.printStackTrace();
}finally{
      try{
            if(stmt!=null)
                  stmt.close();
                  conn.close();
      }catch(SQLException se){
            System.out.println("No se ha podido cerrar la
            conexión.");
      }
}
```

Normalmente, cuando se llama al método que contiene esta parte del código, se le pasará un objeto "Estudiante" con los datos del usuario que nos servirá para ir seteando la sentencia.

2.3.5.2. Actualización de datos de una tabla

La sentencia *UPDATE* es conocida por permitir actualizar valores de una tabla concreta de la base de datos. Recordamos cómo se realiza un *UPDATE:*

```
//Para actualizar datos a la bbdd
UPDATE nombre_tabla SET nombre_columna =valor, nombre_columna
=valor2 WHERE nombre_columna =valor;
```

Este tipo de sentencia se puede ejecutar del mismo modo que en el ejemplo del *INSERT*. Tenemos dos posibilidades: con una sentencia ya preescrita en nuestro código o con datos dinámicos. Así se implementaría:

```
Connection conn = null;
Statement stmt = null;
try {
      // Paso 1: Realizamos la conexión
      //Paso 2. Crear objeto y llamar a la conexión
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");

      //Paso 3. Crear estructura de la sentencia
      String sql = "UPDATE estudiante SET dni = '00000000T'
      WHERE id = '12'";
      //Paso 4. Ejecución
      stmt = conn.createStatement();


}catch(SQLException se){
      //Gestionamos los posibles errores que puedan surgir
durante la ejecucion de la insercion
      se.printStackTrace();
}catch(Exception e){
      //Gestionamos los posibles errores
      e.printStackTrace();
}finally{
      //Paso 5. Cerrar objetos abiertos
      try{
            if(stmt!=null)
                  stmt.close();
                  conn.close();
      }catch(SQLException se){
            System.out.println("No se ha podido cerrar la
conexión.");
      }
}
```

Para un *UPDATE* con sentencia fija, usaremos el objeto *Statement*, ya que es una operación sencilla. El procedimiento es exactamente igual que con otras sentencias: deberemos meter todo el código entre el *try-catch* para poder controlar los errores que puedan surgir. Para un *UPDATE* con datos dinámicos usaremos un objeto *PreparedStatement*, que nos va a permitir añadir dinámicamente los valores que nos interesen a nuestra sentencia.

```
Connection conn = null;
PreparedStatement stmt = null;
try {
// Paso 1: Realizamos la conexion
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");
//Paso 2. Crear estructura de la sentencia, ponemos interrogantes
a los datos que serán dinamicos
      String sql = "UPDATE estudiante SET dni = ? WHERE id = ?";
//Estos datos en una aplicación iran variando, ahora los seteamos
para que contenga valor
      Estudiante estudiante = new Estudiante();
      estudiante.setId(1);
      estudiante.setDni("00000000T");
//Prepararemos la query para que coja los datos de manera
dinamica.
      stmt = conn.prepareStatement(sql);
// Paso 4. Asignamos los valores del objeto que queramos guardar
      stmt.setString(1, estudiante.getDni());
      stmt.setInt(2, estudiante.getId());

//Paso 5. Ejecución
      stmt.execute();

}catch(SQLException se){
      //Gestionamos los posibles errores que puedan surgir
durante la ejecucion de la insercion
      se.printStackTrace();
}catch(Exception e){
      //Gestionamos los posibles errores
            e.printStackTrace();
}finally{
//Cerramos la connexion
      try{
            if(stmt!=null)
                  stmt.close();
                  conn.close();
      }catch(SQLException se){
            System.out.println("No se ha podido cerrar la
conexión.");
      }
}
```

2.3.5.3. Eliminación de datos de una tabla

La sentencia *DELETE* nos servirá para eliminar registros de nuestra base de datos, ya sea porque no queremos que estén o porque ya no los necesitemos. La sentencia se realizaría siguiendo esta estructura:

```
//Para borrar datos a la bbdd
DELETE FROM nombre_tabla WHERE nombre_columna =valor;
```

Cuando realizamos un *DELETE*, siempre tenemos que poner un *WHERE* para poder filtrar mejor los datos que queremos borrar de la tabla, si no es así, se borrarán todos los registros.

También tenemos la posibilidad de realizar la eliminación de los dos modos explicados anteriormente. Aquí vemos dos ejemplos:

```
Connection conn = null;
Statement stmt = null;
try {
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");

      stmt = conn.createStatement();

      String sql = "DELETE FROM estudiante  WHERE dni =
'00000000T'";
      stmt.executeUpdate(sql);
} catch (SQLException e) {
      e.printStackTrace();
} catch (Exception e) {
      e.printStackTrace();
} finally {
      try {
            stmt.close();
      } catch (SQLException e) {
            e.printStackTrace();
      }
}
```

Al ejecutar la sentencia, si no borra ningún registro porque no encuentra el DNI no se producirá ningún error, simplemente no borrará ningún registro. En este caso, también debemos englobar todo el código entre un *try-catch* para poder controlar los errores que se puedan generar.

```
PreparedStatement sentencia = null;
try {
      // Paso 1. Previamente habremos realizado la conexion
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");
      // Paso 2. Crear estructura de la sentencia, ponemos
      interrogantes a los datos que serán dinámicos

      String consulta = "DELETE FROM estudiante  WHERE dni =?"
      // Paso 3. Pasamos la consulta al objeto PreparedStatement
      sentencia = conn.prepareStatement(consulta);

      // Paso 4. Asignamos los valores del objeto que queramos
guardar
      // Imaginemos que el objeto estudiante esta ya declarado y
con datos.

      Estudiante estudiante = new Estudiante();
      sentencia.setString(1, estudiante.getDni());
      // Paso 5. Ejecucion
      sentencia.execute();
} catch (SQLException e) {
                  System.out.println(e.getCause());
}catch (Exception e) {
                  System.out.println(e.getCause());
} finally {
      // Paso 6. cerramos la conexión y el objeto en uso
      try {
            sentencia.close();
      } catch (SQLException e) {
            e.printStackTrace();
      }
}
```

Este ejemplo es una implementación de un *DELETE* con una sentencia dinámica. El objetivo es borrar los registros de la tabla "Estudiante" que tenga como DNI el valor que contenga el *estudiante.getDni()*. El valor será dinámico porque cada objeto "Estudiante" tendrá un valor distinto.

Podréis ver cómo realizar este tipo de consultas en los ejercicios de GitLab.

2.3.6. Ejecución de consultas

Una de las opciones que tenemos para interactuar con las bases de datos son las consultas, también conocidas como *SELECT*. Es una de las sentencias más utilizadas, ya que nos permitirá seleccionar datos según los criterios que le indiquemos. La estructura básica de una consulta debería ser así:

```
//Para borrar datos a la bbdd
SELECT * FROM nombre_tabla WHERE nombre_columna =valor;
```

Desde JDBC tenemos múltiples opciones para poder realizar consultas, tanto una consulta ya preestablecida en el código como una consulta dinámica que irá cambiando a medida que cambien los datos que le indiquemos.

```
Connection conn = null;
Statement stmt = null;
try {
	conn = conector.conector();
	System.out.println("Nos hemos conectado a la BBDD");
	String sql = "SELECT * FROM Estudiante";
	stmt = conn.createStatement();
	ResultSet rs = stmt.executeQuery(sql);
	while (rs.next()) {
		int id = rs.getInt("id");
		String nombre = rs.getString("nombre");
		String apellido = rs.getString("apellido");
		String dni = rs.getString("dni");
		System.out.print("ID: " + id);
		System.out.print(", Nombre: " + nombre);
		System.out.print(", Apellido: " + apellido);
		System.out.println(", DNI: " + dni);
	}
	rs.close();
} catch (SQLException se) {
	se.printStackTrace();
} catch (Exception e) {
	e.printStackTrace();
} finally {
	try {
		if (stmt != null)
		stmt.close();
	} catch (SQLException se) {
		System.out.println("No se ha podido cerrar la
conexion");
	}
}
```

El procedimiento de implementación es bastante senillo y se asemeja al que ya hemos visto con anterioridad.

Como podemos ver, hay dos opciones para realizar consultas. La primera, simplemente creando una selección genérica de los registros de una tabla. Como en ejemplos anteriores, el primer paso será realizar la conexión con nuestra clase auxiliar para realizar la conexión con la base de datos. Seguidamente, crearemos un *string* con la sentencia de consulta que queramos realizar. En este caso, seleccionaremos todos los registros de la tabla "Estudiante". Para realizar esta acción, llamaremos al método de la conexión *createStatement()*, que devolverá un objeto Statement que usaremos más adelante. En este ejemplo, introduciremos la interfaz *ResultSet*, que se utiliza para almacenar las filas seleccionadas de una tabla y nos va a permitir recorrer la lista y buscar las diferentes columnas de cada registro.

A continuación, para ejecutar la consulta, tendremos que pasarle la SQL al método *executeQuery()*. Una vez realizado, con el método *next()* de la interfaz *ResultSet* podremos recorrer los resultados. El ejercicio se encarga de buscar cada columna e imprimirlo en consola.

Para finalizar, tendremos que tener en cuenta que el código debe englobarse en un *try-catch* para controlar los posibles errores. Además, tendremos que recordar cerrar todos los objetos usados.

La segunda opción de implementación de una consulta consiste en realizar consultas dinámicas con datos variables. Como hemos visto con anterioridad, podemos realizar dicha consulta con la interfaz *PreparedStatement* y seteando posteriormente el dato que nos interese. En este caso, usamos un *PreparedStatement* porque vamos a realizar una consulta dinámica, pero guardaremos el resultado en un *ResultSet*. El objeto *ResultSet* tiene un método que nos permitirá recorrer los resultados uno a uno y tratar los datos obtenidos: ese método es *next()*, el cual ya lo hemos visto en el ejemplo anterior.

Como veremos en el ejemplo, podremos acceder a los datos y tratarlos de la manera que deseemos.

```
Connection conexion = null;
PreparedStatement sentencia = null;
Estudiante estudiante = new Estudiante();
try {
	conexion = conector.conector();
	System.out.println("Nos hemos conectado a la BBDD");
	String consulta = "select * from estudiante where nombre =
? ";
	sentencia = conexion.prepareStatement(consulta);
	sentencia.setString(1, estudiante.getNombre());

	ResultSet rs = sentencia.executeQuery();
	while (rs.next()) {
		int id = rs.getInt("id");
		int dni = rs.getInt("dni");
		String nombre = rs.getString("nombre");
		System.out.print("ID: " + id);
		System.out.print(", DNI: " + dni);
		System.out.print(", NOMBRE: " + nombre);
	}

	rs.close();
	sentencia.close();
} catch (SQLException se) {
	se.printStackTrace();
} catch (Exception e) {
	e.printStackTrace();
} finally {
	conexion.close();

}
```

2.3.7. Ejecución de procedimientos almacenados en la base de datos

En este apartado, os enseñaremos a ejecutar un procedimiento desde una aplicación Java con JDBC.

Un **procedimiento** es un subproceso o un subprograma que podemos crear en nuestra base de datos para que se ejecute cuando el usuario haga una llamada a este procedimiento. Es conocido también como *procedure*.

Un *procedure* tiene un nombre, una lista de parámetros y diferentes operaciones SQL. Este tipo de procedimiento solo es aceptado en bases de datos de tipo relacional. Un procedimiento tiene que seguir esta estructura:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `nombre_bbbdd`.`nombre_procedure` $$
CREATE PROCEDURE `nombre_bbbdd`.`nombre_procedimiento`
(parámetros)
BEGIN
  Seccion_declaracion
  Parte_execucion_operaciones_SQL
END $$;
DELIMITER ;
```

El procedimiento para este ejercicio quedará más o menos así:

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `ILERNA`.`ObtenerDatosEstudiante` $$
CREATE PROCEDURE `ILERNA`.`ObtenerDatosEstudiante`
   (IN ES_ID INT, OUT ES_DNI VARCHAR(255))
BEGIN
   SELECT DNI INTO ES_DNI
   FROM ESTUDIANTE
   WHERE ID = ES_ID;
END $$

DELIMITER ;
```

Crear un *procedure* se parece un poco a crear un método: tenemos que darle un nombre y entre paréntesis hay parámetros. A nuestro procedimiento le daremos el nombre "ObtenerDatosEstudiante" e irá envuelto entre los símbolos de acento, y delante

situaremos también entre acentos el nombre de la BBDD en la que queremos guardarlo junto con un punto.

A continuación, definiremos los parámetros. Cada parámetro irá separado por una coma y se diferencia entre dos tipos: uno *IN* y otro *OUT*. El *IN* significará que ese parámetro se usará dentro del procedimiento y el *OUT* será el valor que devolverá la función y que luego recogeremos.

El primer parámetro es *ES_ID* acompañado de *IN*, eso significa que se usará dentro del *procedure*, y luego indica el tipo, que será un *int*. Con esta información se nos indica que ese *ES_ID* será el identificador único de la fila de ese registro y equivaldrá al identificador del objeto "Estudiante".

```
Connection conn = null;
Statement stmt = null;

try {
      // Paso 1: Realizamos la conexion
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");
      //Paso2: definimos el procedimiento
      String procedure = "DELIMITER $$" +
            "DROP PROCEDURE IF EXISTS
ILERNA`.`ObtenerDatosEstudiante` $$" + "CREATE PROCEDURE
`ILERNA`.`ObtenerDatosEstudiante` " +
      "   (IN ES_ID INT, OUT ES_DNI VARCHAR(255))" +
      "BEGIN" +
      "   SELECT DNI INTO ES_DNI" +
      "   FROM ESTUDIANTE" +
      "   WHERE ID = ES_ID;" +
      "END $$" +
      "DELIMITER ;";
      // Paso 3: Creamos el objeto Statement de la conexion
      stmt = conn.createStatement();
      //Paso 4: ejecutamos la sql
      stmt.execute(procedure);
} catch (SQLException ex) {
      System.out.println("Se ha producido un error en la ejecucion
de la SQL: " + ex.getMessage());
} finally {
      try {
            stmt.close();
      } catch (SQLException ex) {
            System.out.println("Se ha producido un error al cerrar
la conexion: " + ex.getMessage());
      }
}
```

Para crear este procedimiento desde nuestra aplicación, seguiremos los pasos de ejercicios anteriores. Los pasos son los mismos, solo cambiará la SQL, como vemos en el ejemplo de arriba. Una vez creado, ya podemos usarlo en nuestra aplicación.

En primer lugar, tendremos que tener creado un procedimiento en nuestra base de datos para que podamos llamarlo desde nuestra aplicación. En el ejemplo, tenemos una sentencia sencilla de un procedimiento para nuestro objeto "Estudiante" que nos devolverá información almacenada en esa tabla.

```
Connection conn = null;
CallableStatement llamadaProcedure =null;
Estudiante estudiante = new Estudiante();
try {
      // Paso 1: Realizamos la conexiOn
      conn = conector.conector();
      System.out.println("Nos hemos conectado a la BBDD");

      String sql = "{call ObtenerDatosEstudiante (?,?)}";
      // Paso 2: Llamada al procedimiento almacenado, tiene que
existir en la BBDD
      llamadaProcedure = conn.prepareCall(sql);
      //Paso 3: Definimos el parametro OUT
      llamadaProcedure.registerOutParameter(2, Types.VARCHAR);
      //Paso 4: definimos el parametro IN
      llamadaProcedure.setInt(1, estudiante.getId());

      //Paso 5:  Ejecuta el procedimiento almacenado
      boolean resultado = llamadaProcedure.execute();
      if(resultado) {
            ResultSet lista = llamadaProcedure.getResultSet();
            while (lista.next()) {
                  //Obtenemos el DNI
                  String dni = lista.getString("dni");
                  System.out.println(dni);
            }
      }
} catch (SQLException ex) {
      System.out.println("Se ha producido un error en la
ejecucion de la SQL: " + ex.getMessage());
}
```

Para llamar a un procedimiento, usaremos esta sentencia:

```
call nombreProcedimiento (?,?);
```

Los interrogantes equivaldrán a cada parámetro de la función. Si hay más de dos, se pondrán más.

En este ejemplo, usamos un *CallStatement*, que es una interfaz preparada para realizar este tipo de operaciones. Rellenaremos este nuevo objeto con la llamada al método *prepareCall()*, pasándole la SQL creada con la llamada al procedimiento. Tras esto, deberemos pasarle al procedimiento los valores de los parámetros, para lo que usamos el método *registerOutParameter()* para los parámetros *OUT*. En este caso, es nuestro segundo parámetro, por eso indicamos un 2 y el tipo *varchar*. Para definir el valor del parámetro *INT*, haremos un *setInt()* indicando que es el parámetro 1 y el valor del ID del estudiante. A continuación, tendremos que ejecutar la sentencia con el método *execute()*. Si devuelve *true*, guardaremos los resultados en un *ResultSet* para poder obtener los registros. Siempre que exista un resultado, recorreremos la lista, guardaremos el valor en un *string* e imprimiremos el valor por consola.

Es muy importante que, para que este ejercicio funcione, tengamos creado nuestro procedimiento en la base de datos. No es necesario realizarlo mediante la aplicación, se puede crear directamente en la base de datos.

2.3.8. Gestión de transacciones

Las **transacciones** son las acciones que nos permiten controlar cuándo y cómo se aplican los cambios en nuestra base de datos

Las transacciones se encargan de tratar una o más sentencias SQL, y si una de ellas falla, toda la transacción falla.

En JDBC, las transacciones se realizan de manera automática por defecto, pero existe la posibilidad de ejecutarlas de manera manual. Esto nos puede aportar una mejora del rendimiento de la aplicación, manteniendo la integridad del proceso de negocio. Para habilitar las transacciones de manera manual, el objeto *Connect* tiene un método

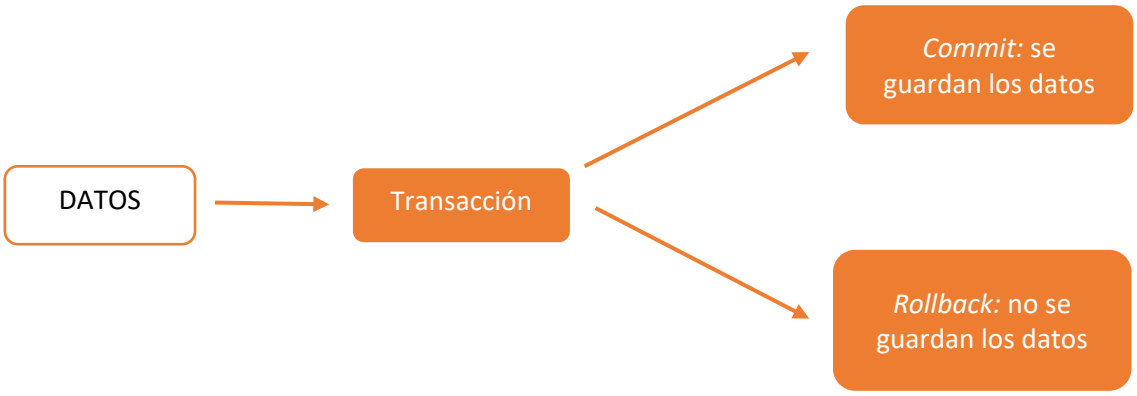


Ilustración 3. Esquema del funcionamiento de las transacciones.

llamado *setAutoCommit()*, al cual tendremos que asignarle el valor *false* para poder establecer las transacciones manuales.

Existen tres métodos del objeto *Connect* que nos serán muy útiles a la hora de realizar transacciones:

- *setAutoCommit(boolean):* permitirá modificar la transacción automática.
- *commit():* efectuará la transacción. Sin la ejecución de este método, no se guardarán los datos en nuestra base de datos.
- *rollback():* es el método que cancela la transacción, nos será útil cuando se produzca alguna excepción.

Para ver cómo se ejecutan, aquí podemos ver un ejemplo:

```
Savepoint savepoint1 = null;

try{
      // Paso 1. Previamente habremos realizado la conexion
      //Paso 2. Creamos un nuevo objeto Statement y estableemos
la transaccion manual
      conn.setAutoCommit(false);
      Statement stmt = conn.createStatement();
      //Paso 3. Creamos un nuevo Savepoint
      savepoint1 = conn.setSavepoint("punto de backup");
      String consulta = "INSERT INTO Estudiante (id, dni, nombre,
apellido, edad) VALUES (22, '11111111H', 'Zara', 'Ali', 18)";
// Paso 4. EjecucIon
      stmt.executeUpdate(consulta);
//Paso 5. Prueba de insert mal hecho
      consulta = "INSERTED IN Estudiante VALUES (107, 22, 'Juan',
'ejemplo')";
      stmt.executeUpdate(consulta);
      // Paso 6. Haremos commit siempre que no se produzca error.
      conn.commit();

}catch(SQLException se){
      conn.rollback(savepoint1);
}
```

Para realizar transacciones manuales, debemos establecer el *autoCommit* a *false*. De esta manera, tendremos control sobre qué se persiste en base de datos y qué no. Si ejecutamos una consulta sin que devuelva error, podremos seguir con el programa. En cambio, si se produce un error, el *commit()* no se realizará. En el ejemplo, ninguno de los *INSERTS* se va a guardar, porque la segunda sentencia está mal construida.

Para evitar perder datos al producirse una excepción, existe la posibilidad de realizar guardados en medio de diferentes sentencias, se los llama *Savepoints*.

```
Savepoint savepoint1 = null;
try{
      // Paso 1. Previamente habremos realizado la conexión
      //Paso 2. Creamos un nuevo objeto Statement y estableemos
la transaccion manual
      conn.setAutoCommit(false);
      Statement stmt = conn.createStatement();
      //Paso 3. Creamos un nuevo Savepoint
      savepoint1 = conn.setSavepoint("punto de backup");

      String consulta = "INSERT INTO Estudiante (id, dni, nombre,
apellido, edad) VALUES (22, '11111111H', 'Zara', 'Ali', 18)";
      // Paso 4. Ejecución
      stmt.executeUpdate(consulta);

      //Paso 5. Prueba de insert mal hecho
      consulta = "INSERTED IN Estudiante VALUES (107, 22, 'Juan',
'ejemplo')";
      stmt.executeUpdate(consulta);
      // Paso 6. Haremos commit siempre que no se produzca error.
      conn.commit();

}catch(SQLException se){
      // Controlamos error y si se produce vuelve al punto de
guardado
      conn.rollback(savepoint1);
}
```

Un ***savepoint*** se encarga de definir un *rollback* lógico en medio de una transacción. Si se produce un error pasado el *savepoint*, podremos ejecutar el método *rollback()* para deshacer todos los cambios o solo los que se encuentran después de este punto de guardado. Podemos destacar tres métodos que nos serán de gran utilidad:

- *setSavepoint(String nombre):* permite definir un punto de guardado nuevo.
- *releaseSavepont(Savepoint nombre):* permite borrar un punto de guardado.
- *rollback(String nombre):* nos permitirá tirar atrás todos los cambios realizados antes de que se produjera un error.

Como vemos en el ejemplo, no se realizará ningún *INSERT* en este caso porque tenemos la segunda consulta mal construida y el punto de guardado está antes de que se ejecuten los *INSERTS*. Para que sea realmente útil, el *savepoint* debería establecerse después del paso 4.

3. Herramientas de mapeo objeto-relacional (ORM)

En este tema introduciremos conceptos importantes sobre la programación tales como el concepto de la programación orientada a objetos, el mapeo objeto-relacional y las características de las herramientas ORM.

Hasta ahora solo habíamos tenido contacto con bases de datos a través del sistema de conexión con la base de datos de tipo relacional JDBC que vimos en el tema pasado. Este tipo de sistema puede implicar una gran cantidad de código para cubrir las necesidades básicas de la aplicación. Las herramientas ORM que introduciremos en este tema nos ofrecen la minimización de código para conseguir el mismo objetivo.

3.1. Concepto de mapeo objeto-relacional (ORM)

El **mapeo objeto-relacional** es una técnica que se utiliza en muchos lenguajes de programación orientada a objetos. Este método facilita la conversión de los datos de los objetos a los atributos de las tablas sin tener que realizar ninguna de las operaciones en nuestra base de datos.

Es decir, para emparejar tu programa con la base de datos es necesario un intermediario que ayude a realizar dicha conexión, a eso se le llama comúnmente como **mapeo**. Para realizar este emparejamiento usaremos los ORM.

La característica principal de las herramientas ORM es que principalmente trabajan con objetos, por lo tanto, toda la metodología seguida depende del paradigma orientado a objetos.

Los ORM generan objetos que se asignan virtualmente a tablas en la base de datos. Una vez que estos objetos están activos, los codificadores pueden trabajar fácilmente para recuperar, manipular o eliminar cualquier campo de la tabla sin prestar mucha atención al lenguaje específicamente. Además, simplifica las consultas SQL más complicadas. Para ello, utiliza librerías para entender el código que estamos programando en forma de objetos y luego lo asigna en la base de datos.

Para realizar el mapeo, según qué lenguaje de programación se use, se utilizará una herramienta u otra.

Este tipo de mapeo es característico de la programación orientada a objetos, y los lenguajes más conocidos que usan este tipo de herramienta son Java, PHP, Python, Ruby y .Net, ya que son lenguajes orientados a objetos.

Esta herramienta será la encargada de coger todos los atributos del objeto que queramos persistir en base de datos, identificará de qué tipo de atributo se trata y lo guardará en su correspondiente columna en la tabla que esté indicada en este mapeo.

Ilustración 35. Esquema concepto ORM.

Este tipo de herramientas se encargará de realizar cualquier tipo de interacción con la base de datos, como hacer *insert*, *select*, *update* o *delete*, sin necesidad de construir ninguna consulta. Gracias a eso, reduciremos las líneas de código y el proceso será mucho más ágil.

3.1.1. Características de las herramientas ORM. Herramientas ORM más utilizadas

La herramienta ORM, tal y como se ha explicado en el punto anterior, será el intermediario entre nuestro programa y la base de datos. Las principales características de este tipo de herramienta son:

- **Seguridad**: es una de las características más importantes, ya que permite proteger los datos y mantener la privacidad. El acceso a esos datos será a través de la herramienta ORM y será impenetrable de otro modo.
- **Agilidad y rapidez**: ayuda a gestionar automáticamente las transacciones entre el objeto y la base de datos sin tener que realizar ningún tipo de operación con la base de datos. Agiliza la construcción del código porque nos evita tener que realizar consultas SQL. Los programadores no necesitan tener nociones de SQL para poder gestionar una herramienta ORM.
- **Abstracción**: es una metodología que permite crear estructuras en la base de datos que nos permitirán seleccionar los datos necesarios sin tener la necesidad de conocer los detalles de esa información.
- **Independiente**: se trata de una aplicación independiente de la base de datos. En caso de migrar a otra base de datos, es bastante bueno tener ORM implementado en el proyecto.

- **Menos restricción de datos**: los cambios en base de datos se pueden realizar mediante la herramienta ORM, permitiendo ser menos restrictivo a la hora de tratar los datos si lo comparamos con JDBC.
- **Robustez**: la conexión con la base de datos se convierte en mucho más robusta, ya que tenemos mucho menos código de por medio. A través del ORM, podremos realizar todas las configuraciones necesarias para mapear los objetos Java. Además, gracias a ello, nuestra aplicación será mucho más segura.

Viendo las características y los beneficios de un ORM, podemos destacar las principales ventajas:

1. Los desarrolladores no necesitan tener conocimientos de SQL para gestionar una herramienta ORM.
2. Promueve la abstracción para mejorar la seguridad de los datos.
3. Permite guardar los procedimientos en la aplicación, permitiendo mucha más flexibilidad para hacer cambios.
4. Una herramienta ORM es muy útil para bases de datos relacionales.
5. Se encarga de realizar todas las operaciones con la base de datos, por tanto, no es necesario codificar las sentencias SQL.
6. Las consultas a través de ORM se pueden escribir independientemente de la base de datos que se esté utilizando, lo que proporciona mucha flexibilidad al codificador. Esta es una de las mayores ventajas que ofrecen los ORM.
7. Los ORM están disponibles para cualquier lenguaje orientado a objetos, por lo que no solo son específicos de un idioma.

Según el lenguaje de programación, utilizaremos un ORM u otro. Los más conocidos son:

- Para lenguaje **Java** tenemos:
  - **Hibernate**: es el más conocido, y es de *software* libre. Se caracteriza por un mapeo a base de XML que ayuda a convertir los datos de los objetos Java definidos a las tablas y atributos correspondientes en base de datos. También es conocido por tener un lenguaje propio de generación de consultas de base de datos.
  - **iBatis**: también es un *framework* de código abierto, menos conocido que el anterior. Está basado en Apache Software Foundation y, como Hibernate, utiliza

ficheros XML para definir la configuración que convertirá los datos de los objetos en sentencias SQL para hacer más sencilla la conexión con la base de datos.

- **Enterprise JavaBeans**: se trata de una API de Java bastante conocida que se utiliza en la construcción de aplicaciones J2EE.

- Para **Pyhton**:
  - **Django**: es una herramienta de código abierto para el desarrollo de aplicaciones que sigan con el modelo MVC (modelo-vista-controlador). Es una herramienta ORM muy conocida y utilizada en este lenguaje.
  - **Storm**: otro ejemplo de herramienta ORM para Python que utiliza lenguaje SQL.
- **PHP**:
  - **Laravel**: es uno de los *frameworks* más conocidos para el desarrollo de aplicaciones PHP. Es gratuito y está basado en Symphony.
  - **Yii**: otro ejemplo de *framework* para el desarrollo PHP, basado en el modelo típico de la programación orientada a objetos, el modelo MVC. Es uno de los *frameworks* más utilizado por ser muy rápido.

3.2. Instalación de una herramienta ORM

En este apartado realizaremos un ejemplo de instalación de herramienta ORM. En este caso, utilizaremos el lenguaje Java e instalaremos Hibernate, una de las herramientas más utilizadas en las empresas y por los programadores.

Hibernate es un ORM pensado para el mapeo objeto-relacional para Java que sigue la arquitectura MVC.

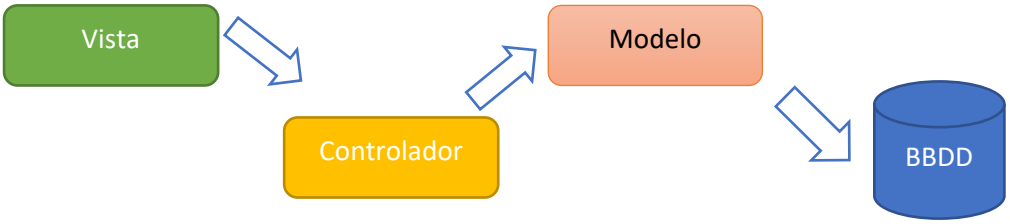


Ilustración 36. Esquema MVC.

La **arquitectura modelo-vista-controlador** (MVC) es un patrón de arquitectura de una aplicación que se encarga de separar la aplicación en tres partes: la vista, el controlador y el modelo de datos.

Este tipo de arquitectura se pensó para separar la información que se transmite al usuario y la que se guarda en la base de datos. La vista será la parte que el usuario puede ver; el controlador, la parte del código que hará de intermediario entre la vista y el modelo; y el modelo se encargará de guardar los datos en la base de datos.

Para empezar, tenemos que crear un proyecto nuevo en Eclipse. Si no tenemos instalado Eclipse, debemos instalar la última versión disponible, en este caso, la versión Eclipse IDE 2019-12. Al descargarla, instalaremos la versión que indica la captura.

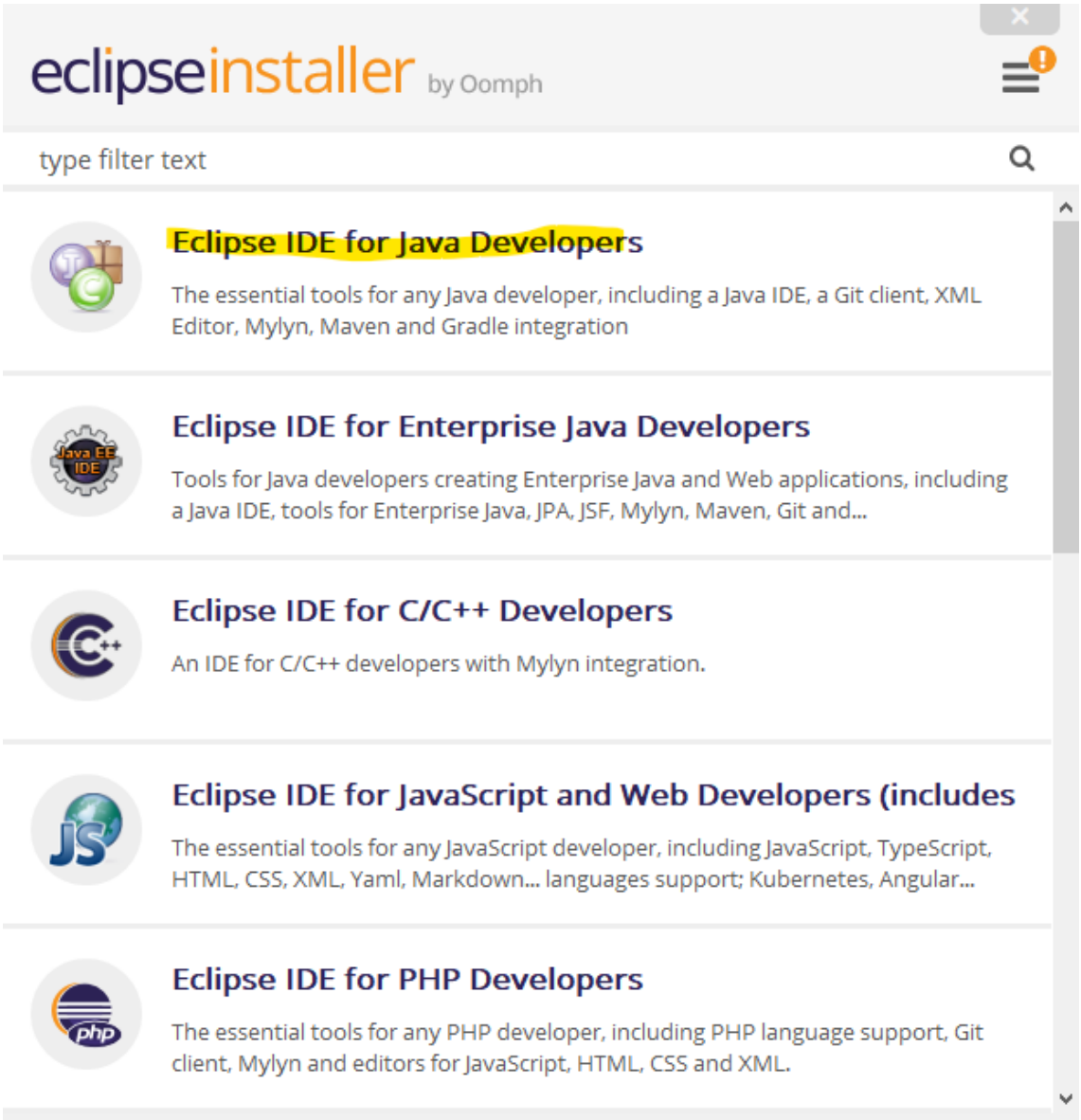


Ilustración 37. Captura del instalador de Eclipse.

1. Tenemos que crear un nuevo proyecto Maven (si no estáis familiarizados con Maven, es una herramienta de automatización para proyectos Java. Profundizaremos en ello más adelante).

   Para crear un nuevo proyecto, iremos a *File > New > Maven Project*.

   > Para los que no sepan crear un Maven Project, pueden ver un tutorial en este enlace:
   >
   > https://dzone.com/articles/how-to-create-a-web-project-using-maven-in-eclipse-1)

   No es necesario utilizar un proyecto Maven para poder añadir la librería de Hibernate, con un proyecto Java sería suficiente.

2. Procederemos a la instalación de Hibernate. Se puede realizar de dos modos: añadiendo manualmente los .jar de Hibernate al proyecto o añadiendo las dependencias al archivo pom.xml del proyecto que hemos generado. El fichero pom.xml es el fichero XML que contiene la información y la estructura del proyecto. Es uno de los ficheros de configuración más importantes porque es la base del proyecto a la que se le irán añadiendo las dependencias necesarias para ir construyendo nuestra aplicación. Las dependencias no son más que las librerías que nos ayudarán a construir nuestro proyecto.

   En este caso, necesitamos las librerías de Hibernate. Utilizaremos la versión que más nos convenga y lo podemos descargar de este enlace, pero es importante revisar si hay una versión superior que nos interese más si es conveniente:

   > https://mvnrepository.com/artifact/org.hibernate/hibernate-core/5.4.14.Final

3. Una vez descargados, añadiremos los .jar a nuestro proyecto. Para ello, deberemos dirigirnos a nuestro proyecto, clicar con el botón derecho y dirigirnos a la opción *Properties*. Veremos una ventana con diferentes opciones, nos dirigiremos a *Java Build Path* y añadiremos nuestras librerías con la opción a la derecha de *Add External JARs*.

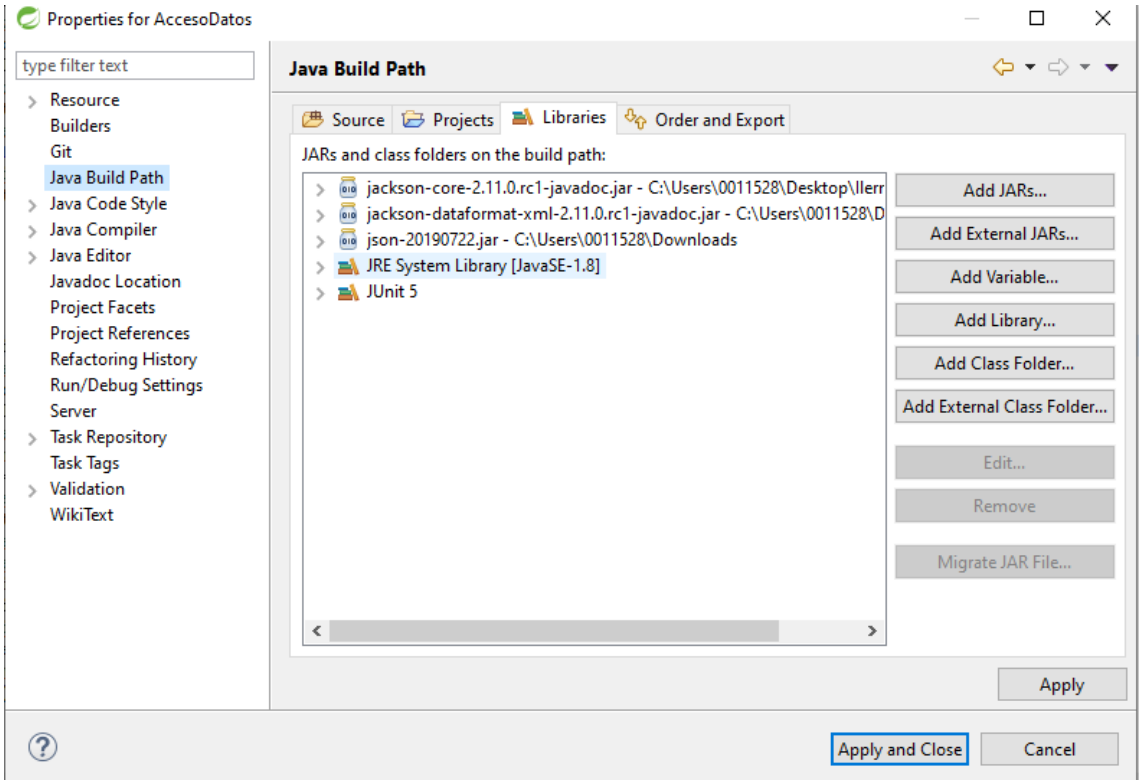


Ilustración 38. Ventana de Eclipse para añadir las librerías de Hibernate.

Una vez terminado, pulsamos *Apply and Close* y ya tendremos las librerías en nuestro proyecto.

Tal y como hemos contado en el tema anterior, deberemos tener configurada la base de datos en nuestro proyecto. Podéis seguir las instrucciones dadas en el tema 2.

3.3. Estructura de un fichero de mapeo. Elementos, propiedades mapeo de colecciones, relaciones y herencia

En este apartado explicaremos cómo realizar la vinculación de nuestra aplicación con la base de datos. Para que Hibernate funcione, necesita que cada clase que queramos persistir tenga un fichero XML que indicará qué tabla y columna corresponde.

Para poder configurar y hacer que funcione Hibernate en nuestra aplicación, será preciso realizar diferentes pasos aparte de añadir librerías. En ese apartado explicaremos paso a paso los ficheros que necesitaremos para que nuestro ORM funcione correctamente.

Para empezar a configurar Hibernate, crearemos estas tres carpetas dentro de nuestro proyecto:

src/main/java (esta solo se creará si no tenemos clases Java ya creadas)

src/main/resources

src/main/webapp

Es importante tener una organización dentro de nuestro proyecto, para tener claro dónde está cada fichero.

Deberíamos tener algo así:

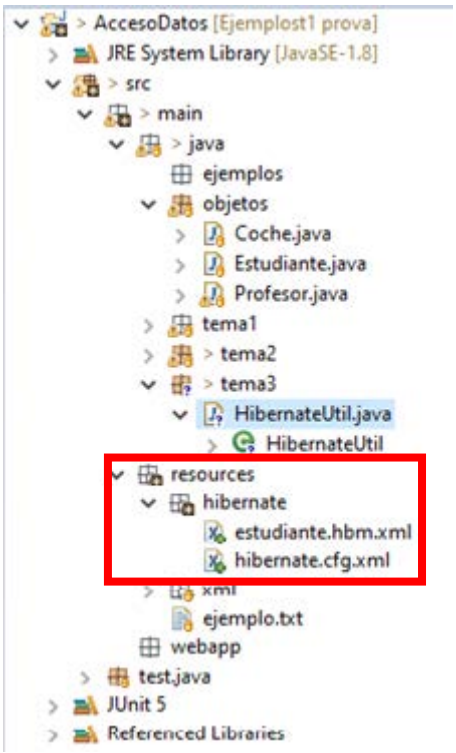


Ilustración 39. Estructura de carpetas de nuestro proyecto.

Dentro de la carpeta Java, tenemos todas las clases de nuestro proyecto. En este caso, os enseñaremos a guardar la información de la clase "Estudiante" que ya tenemos

creada. En la clase no tendremos que poner nada adicional, solo tenemos que comprobar que existan todos los atributos que equivalen a la tabla de la base de datos.

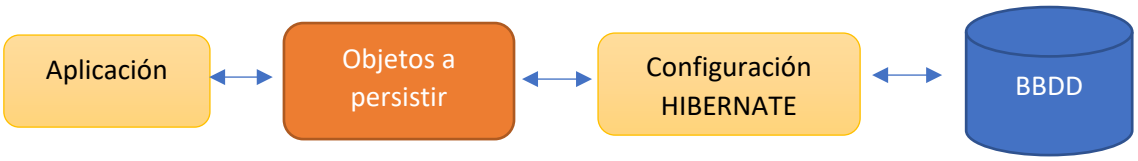


Ilustración 40. Esquema de cómo interactúan los ficheros Hibernate entre la aplicación y la base de datos.

A continuación, tenemos que crear nuestro fichero de configuración de Hibernate. La configuración de Hibernate se realiza mediante ficheros XML que integraremos dentro de nuestro proyecto. Estos archivos son muy importantes, porque nuestro ORM necesita saber cómo debe cargar y guardar la información de la base de datos. Básicamente, los archivos de configuración, contendrán una relación entre la clase Java y la base de datos, y una definición de la conexión entre esta base de datos y nuestra aplicación.

Podemos distinguir dos tipos de ficheros de configuración:

- ***Mapping* .hbm.xml**: son los archivos por cada clase Java que tengamos y su relación con la base de datos. Básicamente, es la traducción entre la clase Java y cómo corresponderá en la base de datos. El nombre del fichero debe corresponder a la tabla que se representa. Por ejemplo, si necesitamos mapear la tabla "Estudiante", el fichero .hbm se llamará estudiante.hbm.xml.
- **Hibernate.cfg.xml**: es el fichero que contendrá todos los ficheros hbm.xml de cada clase Java y la conexión con la base de datos.
- Existe otra opción de mapeo de los objetos Java que consiste en **anotaciones** de Hibernate para relacionar el objeto con la base de datos, ahorrándonos el fichero .hbm.xml.
- **Clase auxiliar *HibernateSession*:** es una clase auxiliar que crearemos para obtener la sesión de Hibernate. En esta clase le indicaremos dónde tenemos los anteriores archivos de configuración.

3.3.1. *Mapping* .hbm.xml

Para empezar, veremos cómo crear un fichero de mapeo. Se trata de un fichero XML que seguirá la estructura explicada en el tema anterior, pero además añadiremos la declaración del tipo de documento:

```
<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-mapping
  PUBLIC '-//Hibernate/Hibernate Mapping DTD 5.3//EN'
  'http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd'>

<hibernate-mapping>
      <class name="nombrePropedadOBjeto"
table="Nombre_tabla_BBDD">
            <id name="id" type="integer">
                  <generator class="assigned"/>
            </id>
            <property column="nombre_propiedad_Objeto"
            name="nombre_Columna BBDD" type="tipo_dato"/>
      </class>
</hibernate-mapping>
```

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
<hibernate-mapping>
      <class name="main.java.objetos.Estudiante" table="estudiante">
            <id name="id" type="integer">
                  <generator class="assigned"></generator>
            </id>
            <property name="dni" column="dni" type="string"/>
            <property name="nombre" column="nombre" type="string" />
            <property name="apellido" column="apellido" type="string"/>
            <property name="edad" column="edad" type="integer"/>
      </class>
</hibernate-mapping>
```

Se trata de la declaración del tipo de documento XML.DOCTYPE que nos indica el tipo de documento que debe relacionarse con este XML, es decir, el modelo que hemos de utilizar para marcar este texto. Por eso, añade información y una URL del modelo en cuestión que se llama hibernate-mapping-5.3.dtd. Añadir esta información nos ayudará a dotar al XML del modelo de Hibernate para poder añadir las etiquetas y validarlo.

Este fichero deberá seguir una estructura:

1. Primero definiremos la cabecera de un fichero típico de un XML.
2. A continuación, añadiremos la declaración del DOCTYPE añadiendo la referencia de Hibernate. Si añadimos esta referencia, podremos saber todas las etiquetas permitidas en este tipo de fichero.
3. Para empezar a definir el fichero de mapeo, deberemos meter la etiqueta *hibernate-mapping*, la cual contendrá toda la parametrización del XML.
4. Dentro definiremos la etiqueta *class*, que contendrá el nombre de la clase Java y su equivalencia en la base de datos. El nombre de la clase debe corresponder al nombre del paquete en que se encuentra. Dentro de esta etiqueta iremos metiendo los atributos de nuestra clase Java.
5. Continuaremos con el mapeo de la clave primaria. Utilizaremos la etiqueta *id*, a la cual le pondremos un *name* que corresponderá al nombre de la propiedad Java a la que equivale la clave primaria. El atributo *column* nos servirá para establecer el nombre de la columna equivalente de la base de datos. También deberá tener un *type* con el tipo, aunque no es necesario porque Hibernate usa el tipo de la propiedad Java. Como vemos, dentro de la etiqueta *id* añadiremos la etiqueta *generator*, con una *class* que podrá tener diferentes valores según nuestras necesidades. Los tipos que podemos encontrar son estos:

Valor	Descripción
assigned	Es la estrategia por defecto, se encargará de asignar un *id* por defecto.
increment	Se encarga de generar un *id* único, que se irá incrementando. El tipo de este *id* puede ser *short*, *int* o *long*.
sequence	Utilizará la secuencia de la base de datos. Si la base de datos no tiene secuencia, creará una automáticamente. Si la base de datos tiene secuencia, tendremos que añadir el nombre de la secuencia dentro de la etiqueta *generator*, por ejemplo: <param name="sequence">nombre_secuencia</param>
hilo	Se encarga de generar el *id* mediante un logaritmo, el tipo será *short*, *int* o *long*.

native	Se encarga de elegir el tipo *identify*, *sequence* o *hilo* dependiendo del tipo de base de datos que utilicemos.

Debemos tener en cuenta que la relación de tipos permitidos es esta:

Tipo mapping	Tipo Java	Tipo SQL
integer	int o java.lang.Integer	INTEGER
long	long o java.lang.Long	BIGINT
short	short o java.lang.Short	SMALLINT
float	float o java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte o java.lang.Byte	TINYINT
boolean	boolean o java.lang.Boolean	BIT
date	java.util.Date o java.sql.Date	DATE

Tabla 14. Tipos y su relación con el objeto Java y SQL.

El tipo que tendremos que usar es el tipo *mapping*, que es el que corresponde a este tipo de archivo.

6. Para el resto de atributos de la clase, utilizaremos la etiqueta *property*, donde pondremos los atributos *name* para el nombre Java, *column* para el nombre correspondiente de la columna en la base de datos y un *type* para el tipo de atributo.

Puede parecer complicado, pero realmente es sencillo. Vamos a ver un ejemplo práctico de cómo debería quedar este fichero según nuestra clase Java "Estudiante".

Este fichero lo meteremos dentro de la carpeta creada anteriormente: **src/main/resources**.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration
  PUBLIC '-//Hibernate/Hibernate Configuration DTD 5.3//EN'
  'http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd'>
<hibernate-configuration>
      <session-factory>
            <property name="hbm2ddl.auto">update</property>
            <property name="dialect">
org.hibernate.dialect.MySQL5Dialect</property>
            <property
name="connection.url">jdbc:mysql://localhost/ilerna</property>
            <property name="connection.username">usuario</property>
            <property name="connection.password">password</property>
            <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
            <mapping resource="estudiante.hbm.xml"/>
      </session-factory>
</hibernate-configuration>
```

3.3.2. Archivos de configuración hibernate.cfg.xml.

Otro fichero importante que nos servirá para configurar la conexión entre la base de datos e Hibernate es el fichero hibernate.cfg.xml.

Este fichero presenta una estructura parecida al anterior. Se define como un documento XML, con la declaración de DOCTYPE. El *tag* raíz es diferente, en este caso, usa *<hibernate-configuration>*.

Dentro de ese *tag* añadiremos *<session-factory>*, que contendrá la configuración de Hibernate. Las siguientes líneas contendrán la configuración de Hibernate en el *tag <property>*. El atributo *name* contiene la propiedad de la configuración.

Estas son las propiedades que se usan para conectarse:

Propiedad de configuración	Descripción
connection.url	La URL de la base de datos que vamos a usar.
connection.username	El usuario de conexión a la BBDD.

connection.password	La contraseña de la BBDD.
dialect	Propiedad opcional. Indicará qué lenguaje SQL se va a usar con Hibernate. Podemos encontrar la lista de dialectos que soporta Hibernate en el paquete org.hibernate.dialect.
hibernate.show_sql	Propiedad opcional. Sirve para indicar si se mostrará por consola la SQL que Hibernate lanza. Valores *true* o *false*. Es útil para ver si nuestra consulta está funcionando.
connection.datasource	Sirve para indicar el nombre del *datasource* con el que se conectará Hibernate a la base de datos.

Tabla 15. Propiedades que se usan en el fichero de configuración de Hibernate.

3.3.3. Configurar objetos con anotaciones de Hibernate

En el apartado 3.3.1 os hemos explicado un método para mapear la clase Java con un fichero XML. En este apartado os explicaremos otra manera de realizar ese paso, pero con anotaciones directamente en la clase Java que persistir.

Las **anotaciones** son etiquetas que se añaden a la clase y a los atributos para indicar qué relación tienen con la base de datos.

La utilización de este tipo de anotaciones nos ahorrará crear el fichero .hbm.xml de la clase que estemos mapeando. Para ello, será necesario añadir esta librería a nuestro proyecto:

https://repo1.maven.org/maven2/org/hibernate/javax/persistence/hibernate-jpa-2.1-api/1.0.2.Final/hibernate-jpa-2.1-api-1.0.2.Final.jar

Las anotaciones se pueden clasificar en tres grupos:

3.3.3.1. Anotaciones de entidad

Una entidad hará referencia a la clase que se quiere persistir en la base de datos. También la podemos nombrar como clase entidad.

Estas etiquetas se usan para para mapear la entidad con la base de datos. Las más importantes son:

TAG	Descripción
@Entity	Se utiliza para marcar la clase como una entidad de Hibernate que se tiene que persistir en la base de datos.
@Table	Se utiliza para definir la equivalencia con la tabla de BBDD.
@Id	Sirve para indicar la clave única de la base de datos, es decir, la *primary key*.
@GeneratedValue	Esta etiqueta se utiliza para definir que el campo se generará automáticamente.
@Column	Se utiliza para mapear el campo con el atributo de la tabla de la BBDD, se puede definir la longitud del campo o si es obligatorio o no.
@OrderBy	Se usa para indicar que esa columna se ordenará del modo que se indique. El orden puede ser *asc* o *desc*.
@Transient	Se utiliza para indicar que ese atributo no se guarda en la base de datos.

Tabla 16. Tags más importantes para el mapeo de una clase persistente.

Como podemos ver en el ejemplo, lo primero será añadir la etiqueta *@Entity*, que indicará a Hibernate que esa clase se guardará en la base de datos. La etiqueta *@Table* se usará para indicar a qué tabla de la base de datos corresponde. Estas etiquetas se añaden a la clase.

En los atributos también deberemos añadir etiquetas. En primer lugar, debemos asegurarnos de que nuestra clase tendrá un atributo *@id*, que garantizará que sea un registro único. Tendremos que añadir la etiqueta *@Id* y *@GeneratedValue*, para indicar que se trata de un identificador único, además de la etiqueta *@Column*, ya que esta etiqueta servirá para indicar a que columna de la tabla corresponde.

Cada atributo de la clase deberá tener una etiqueta *@Column*.

```
@Entity
@Table (name="profesor")
public class Profesor {
	@Id @GeneratedValue
	@Column (name="id")
	private int id;
	@Column (name="dni")
	@OrderBy("desc")
	private String dni;
	@Column(name = "nombre")
	private String nombre;
	@Column(name = "apellido")
	private String apellido;
	@Column(name = "edad")
	private int edad;

}
```

3.3.3.2. Anotaciones entre tablas

Este tipo de anotaciones se usa para establecer relaciones entre tablas. En ocasiones, en las entidades se define como atributo un objeto, y este objeto tiene una relación con otra tabla, por lo que debemos definir una relación.

Estas anotaciones se añadirán a los atributos.

Tenemos otro conjunto de anotaciones que se usan para especificar la relación entre columnas y entre diferentes tablas y entidades.

Tag	Descripción
@OneToOne	Se usa para establecer la relación uno a uno.
@OneToMany	Se usa para establecer la relación de ese atributo con múltiples tablas.
@PrimaryKeyJoinColumn	Esta anotación se utiliza para asociar entidades que comparten la misma clave primaria.
@JoinColumn	Se usa para asociaciones uno a uno o muchos a uno cuando una de las entidades posee una clave foránea.
@JoinTable	Se utiliza para entidades vinculadas a través de una tabla de asociación.

@MapsId	Se usa para persistir dos entidades con clave compartida.
@ManyToMany	Se utiliza cuando la relación entre tablas es de muchos a muchos. Por ejemplo, un estudiante puede elegir muchas asignaturas, y las asignaturas pueden tener muchos estudiantes.

Tabla 17. Anotaciones entre tablas.

```
@Entity
@Table(name = "pregunta")
public class Entrevista {

	@Id
	@GeneratedValue(strategy = GenerationType.TABLE)
	private int id;
	private String nombrePregunta;

	@OneToMany(cascade = CascadeType.ALL)
	@JoinColumn(name = "id")
	@OrderColumn(name = "type")
	private List<Preguntas> preguntas;
	//constructor,getter, setter
}
```

Como vemos en el ejemplo, tenemos una lista de preguntas que tiene la etiqueta *@OneToMany:* lo que indicará esta anotación es que cada clase "Entrevista" tendrá una lista de preguntas y tendrá una relación *OneToMany*, es decir, que la clase podrá tener más de una pregunta.

3.3.3.3. Anotaciones herencia

En Java, la herencia es una de los conceptos principales. El hecho es que transformar esta herencia en mapeo de Hibernate puede resultar un problema porque las bases de datos relacionales no cuentan con un proceso para realizar la operación, es decir, SQL, Hibernate o cualquier otro ORM no soporta este tipo de mapeo. Para poder solucionar este problema, JPA (Java Persistance API) tiene ciertas estrategias para abordar esta problemática.

JPA es una API de persistencia que describe la gestión de datos relacionales en Java y viene desarrollada en la plataforma JAVA EE.

Tag	Descripción
@Inheritance	Se usa para indicar qué estrategia de herencia se utilizará.
@DiscriminatorColumn	Como su nombre indica, esta columna es el discriminador, y esta anotación especifica la columna discriminada para las estrategias de mapeo de herencia *SINGLE_TABLE* y *JOINED*.
@DiscriminatorValue	Se utiliza para indicar que esa clase será filtrada con el valor que se asigne.
@PrimaryKeyJoinColumn	Se utiliza para indicar que esa columna se utiliza para unir con otra a través de un identificador.
@MappedSuperclass	Se utiliza para indicar que esa clase es una clase padre.

Tabla 18. Anotaciones más importantes.

Para poder resolver la problemática, JPA abordó el problema con cuatro soluciones:

Mapeo superclase

Esta estrategia de mapeo es el enfoque más simple para asignar una estructura de herencia a las tablas de la base de datos. Se establece que la clase padre no puede ser una entidad. Por ejemplo, si tenemos una superclase "Persona":

```
@MappedSuperclass
public class Persona {
	@Id
	private long idPersona;
	private String nombre;

  //constructor, getters, setters
}
```

```
@Entity
public class Alumno extends Persona {
	private String dni;

	//constructor, getters, setters
}
```

Tendremos que añadir la anotación *MappedSuperclass*, que indicará que esa clase no se mapea porque es una superclase. En cambio, la clase "Alumno" es una entidad porque extiende de "Persona". Por eso, a esa subclase deberemos añadirle la etiqueta *@Entity*.

En la base de datos, la clase "Alumno" equivaldrá a la tabla "Alumno" y tendrá tres columnas: las dos primeras de la superclase y la de la subclase. Por tanto, la representación en la base de datos será con los datos de la superclase más los de la entidad.

Asigna cada clase concreta a su propia tabla. Eso permite compartir la definición de atributo entre varias entidades, pero también supone un gran inconveniente: el mapeo de una superclase no es una entidad, y no hay una tabla para ello.

Tabla única

Esta estrategia es muy similar a la anterior, la principal diferencia es que ahora la superclase también es una entidad. La estrategia de tabla única crea una tabla para cada jerarquía de clase, la superclase y las subclases estarán dentro de la misma tabla. Esta es también la estrategia predeterminada elegida por JPA si no especificamos una explícitamente. Eso hace que la consulta para una clase específica sea fácil y eficiente.

Esta estrategia se centrará en agrupar las clases padre e hijo en una misma tabla. Dado que los registros de esta jerarquía de clases estarán en la misma tabla, Hibernate necesita una forma de identificar a qué entidad pertenece cada registro.

```
@Entity(name="articulos")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipo_articulo",
  discriminatorType = DiscriminatorType.INTEGER)
public class Articulo{
      //atributos,constructor, getters, setters
}
```

Como podemos apreciar por el ejemplo, la clase padre se mapeará como una entidad y se añadirá la etiqueta *@Inheritance*, definiendo la estrategia como *InheritanceType.SINGLE_TABLE*. Esta definición indicará que la estrategia de herencia será de tabla única y es necesario añadirla a toda clase que sea superclase. Ahora deberemos definir qué columna se utilizará para distinguir a qué clase pertenece cada registro. Para ello utilizamos la etiqueta *@DiscriminatorColumn* y le asignamos un nombre "tipo_articulo". Aquí se añadirá el nombre de la clase y se podrá filtrar cuando se haga una consulta. Si no se especifica el nombre, Hibernate establecerá esa columna como *DTYPE*.

Para las subclases, se añadirá solo la anotación *@Entity* y la anotación *@DiscriminatorValue*.

```java
@Entity
@DiscriminatorValue("1")
public class Lapiz extends Articulo{
      //atributos,constructor, getters, setters
}
```

```java
@Entity
@DiscriminatorValue("2")
public class Libreta extends Articulo{
      //atributos,constructor, getters, setters
}
```

En la definición de las subclases, se debe añadir la anotación *@DiscriminatorValue*, que especificará el valor que discriminar para esa entidad. Es decir, el valor 1 indicará que la clase a la que pertenece ese registro es 1, y el 2 hará referencia a la clase "Libreta". Es una manera simple de identificar la clase a la que pertenecen, pero también se puede indicar que la columna sea un *string* y asignar un valor de texto. En lugar de poner *discriminatorType=DiscriminatorType.INTEGER*, se pondría *discriminatorType=DiscriminatorType.STRING*.

Este tipo de estrategia es una manera eficiente y fácil de acceder a los datos de la base de datos. Todos los atributos de cada entidad se guardan en una misma tabla, y las consultas no necesitan de *joins* para ejecutarse. La única particularidad es que Hibernate necesita añadir a la consulta SQL una comparación del valor discriminador para obtener la entidad.

Joined table

Esta estrategia, a diferencia de la anterior, se encarga de mapear cada clase en una tabla propia. La única columna que se repite es el identificador, que se utilizará para enlazar las tablas.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
    @Id
    private long idAnimal;
    private String especie;

    // constructor, getters, setters
}
```

En la clase padre, deberemos definir la clase con la etiqueta *@Entity* y en *strategy* de la etiqueta *@Inheritance* deberemos añadir *InheritanceType.JOINED*.

Para la subclase se definirá así:

```
@Entity
@PrimaryKeyJoinColum("idGato")
public class Gato extends Animal {
    private String nombre;

    // constructor, getters, setters
}
```

Las dos clases tendrán un identificador "idAnimal". La clave primaria de la entidad "Gato" también tiene una restricción de clave externa para la clave primaria de su clase padre. Para personalizar esta columna, podemos agregar la anotación *@PrimaryKeyJoinColumn* y añadir el nombre de la columna.

La desventaja de este tipo de estrategia es que la recuperación de entidades requiere uniones entre tablas, lo que puede resultar en un menor rendimiento para grandes cantidades de registros. El número de combinaciones es mayor cuando se consulta la clase principal, ya que se unirá con cada elemento secundario relacionado, por lo que es más probable que el rendimiento se vea afectado en la superclase de la que queremos recuperar registros.

3.3.4. Creación clase auxiliar *HibernateSession*

Para finalizar, debemos tener una clase que se encargue de conectar la base de datos con los ficheros de configuración que hemos creado. Para que todo funcione correctamente, vamos a necesitar una clase auxiliar, a la cual podemos llamar *HibernateUtil* o *HibernateSession*, como queramos, que nos va a ayudar a leer el archivo de configuración que hemos creado antes, el hibernate.cfg.xml.

El archivo debería ser algo así:

```
private static SessionFactory conexion() {
        try {
            // Crea la SessionFactory desde el fichero
            hibernate.cfg.xml
            Configuration configuration = new Configuration();

configuration.configure("src/main/resources/hibernate/hibernate.cf
g.xml");
            System.out.println("Se ha cargado la configuración");

            ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder().applySettings(configuration.getPr
operties()).build();

            SessionFactory sessionFactory =
configuration.buildSessionFactory(serviceRegistry);

            return sessionFactory;
        }
        catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation
failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
```

La conexión se puede realizar buscando el fichero de hibernate.cfg.xml o directamente indicando cada propiedad definida en ese fichero y pasando a mano cada configuración. Lo más apropiado es tener aparte la configuración, pero os explicaremos esta otra opción para que sepáis que existe.

Este método se deberá incluir en la clase auxiliar que creemos, y se encarga de crear una nueva instancia de *Configuration*. A través del método *configure()*, le pasaremos la ruta del fichero de configuración. Seguidamente, deberemos crear una nueva instancia de la clase *ServiceRegistry* y le aplicaremos los datos de la configuración obtenidos. Crearemos una *SessionFactory* y con el método *buildSessionFactory()* le pasaremos la información del servicio que registrar y creará una sesión.

Revisaremos este proceso más detalladamente en el apartado 3.7.

3.4. Clases persistentes

El propósito de Hibernate es acceder a los atributos de una clase y poder conservarlos en una tabla de nuestra base de datos. Normalmente, estas clases se llaman clases persistentes.

Podemos entender como **clase persistente** un tipo de clase Java que representará la entidad que queremos persistir en la base de datos.

Como hemos explicado en el apartado anterior, para establecer la relación entre la clase e Hibernate, tendremos que crear un XML que servirá para vincular la clase con la tabla de la base de datos. Este fichero ayudará a Hibernate a establecer cómo extraer la información de los atributos e introducirlos con la tabla y los campos asociados.

En Hibernate, funciona con este tipo de asociación, también conocido como POJO *(plain old Java object)*.

Un **POJO** es una nueva instancia de una clase, la cual no extiende ni implementa nada especial.

Las clases persistentes siguen una serie de reglas que son aconsejables, pero no obligatorias:

- **Identificador**: todas las clases deberían tener un identificador único que las distinga de otro registro en la base de datos.
- **Constructor por defecto**: todas las clases persistentes deberían usar su constructor por defecto.
- Los atributos de la clase deben declararse privados y tener un *getter* y un *setter*.

Vamos a ver un ejemplo de cómo definir una clase persistente:

```
public class Estudiante {
	//Atributos
	private int id;
	private String dni;
	private String nombre;
	private String apellido;
	private int edad;

	//Getter
	public int getId() {
		return id;
	}
	//Setter
	public void setId(int id) {
		this.id = id;
	}

	//Continuación getter/setters otros atributos
}
```

Como podemos apreciar, tenemos una clase "Estudiante" que tiene declarados todos los atributos que se tienen que persistir en la base de datos. También incluye un atributo "id", que nos asegurara que cada objeto sea único. Se definirán como *private* porque el acceso a esos datos tiene que ser privado solo con acceso desde la aplicación. A continuación, cada atributo deberá tener su *getter* y su *setter:* el *getter* es el que se utiliza para recoger los datos de la base de datos; en cambio, el *setter* es el encargado de guardar la información. Cada vez que necesitemos guardar o consultar datos, se harán las llamadas pertinentes a su *getter* o su *setter*. Es una parte muy importante de la clase, ya que sin ella no podremos realizar ninguna de estas acciones.

La clase "Estudiante" no tiene constructor, ya que usará el definido por defecto.

3.5. Sesiones, estados de un objeto

Para establecer una conexión con la base de datos, Hibernate usa la interfaz *Session*, pero ¿qué es una sesión en Hibernate?

Las **sesiones** son una construcción de Hibernate que se utilizan para realizar las conexiones con la base de datos.

Las **sesiones** se utilizan para realizar la conexión con la base de datos. Usaremos esta interfaz para realizar la conexión, abriendo una única conexión de base de datos y manteniéndola abierta hasta su cierre. Cada objeto que Hibernate carga desde la base de datos está asociado con la sesión, lo que permite que Hibernate persista automáticamente en los objetos que se modifican y permite implementar funcionalidades como la carga diferida.

Su función principal es ofrecer, crear, leer y eliminar operaciones para los diferentes objetos e información de la aplicación. La interfaz *Session* se caracteriza por su ligereza y ha sido diseñada para instanciarse cada vez que se tenga que interactuar con la base de datos.

Este tipo de objetos tendrán que mantenerse abiertos solo el tiempo necesario para realizar la tarea encomendada. Una vez finalizada la acción, debe cerrarse el proceso, esto nos ayudará a tener control de los procesos abiertos de nuestra aplicación y a no sobrecargarla.

El ciclo de vida de una sesión está limitado por el principio y el final de una transacción lógica. La función principal de la sesión es ofrecer operaciones de creación, lectura y eliminación para instancias de clases de entidad mapeadas. Pueden existir instancias en uno de tres estados:

- **Persistente *(persistent)***: decimos que una sesión es persistente cuando la información que alberga tiene una representación en la base de datos con un identificador.
- **Transitorio *(transient)***: es un estado que se caracteriza por ser una nueva instancia de una clase persistente, la cual no está asociada con una sesión y no tiene representación en la base de datos ni ningún identificador. Es decir, la información solo se encuentra en la sesión, no en la base de datos.
- **Separado *(detached)***: este tipo de estado de la sesión se define por ser una instancia separada de una sesión que ya se ha guardado en la base de datos, pero la sesión dentro de la aplicación ya se ha cerrado: el objeto *Session* ya no tiene esa información, pero la base de datos sí.

Existen gran cantidad de métodos en la interfaz sesión. Antes de ver un ejemplo, debemos prestar atención a estos métodos y conocer de qué se encargan. Los más importantes son:

Retorna	Método	Descripción
Transaction	beginTransaction()	Este método se usará para empezar una unidad de trabajo y devolverá un objeto de transacción asociado.
void	cancelQuery()	Este método nos ayudará a cancelar la ejecución de una consulta.
void	clear()	Método utilizado para limpiar la sesión.
Criteria	createCriteria(String nombre entidad)	Este método creará una nueva instancia de la clase *Criteria* del objeto que se le pase por parámetro.
Query	createQuery(String consulta)	Crea una nueva instancia de la clase *Query* con la consulta que le pasamos por parámetro.
SQLQuery	createSQLQuery(String consulta)	Crea una nueva instancia de *SQLQuery* con la consulta que le pasemos por *string*.
Connection	close()	Cierra la sesión.
Serializable	getIdentifier(Object object)	Este método se encarga de devolver el identificador de la entidad que se le pasa por parámetro.
void	delete(Object objeto)	Borrará el objeto que le pasemos por parámetro de la base de datos.
Session	get(String nombreEntidad, Serializable id)	Devuelve una instancia de la entidad que le pasamos por *string* y del id correspondiente, o nulo si no existe ningún registro.
SessionFactory	getSessionFactory()	Devuelve un *SessionFactory* que ha creado la sesión.

Serializable	save()	Método que guardará el objeto en la base de datos.
void	saveOrUpdate(Object objeto)	Método que se encargará de guardar o actualizar el objeto que le pasemos por parámetro en la base de datos.
void	update(Object objeto)	Este método se encarga de actualizar la entidad que le indiquemos en la base de datos.

Tabla 19. Métodos más importantes de la interfaz Session.

Estos son los métodos más usados por el objeto *Session*. Este sería un ejemplo de uso de uno de estos métodos.

```
util = new HibernateUtil();
factory = util.getSessionFactory();
Session session = factory.openSession();
Transaction tx = null;
try {
       tx = session.beginTransaction();
       // más codigo

       tx.commit();
    }catch (Exception e) {
       if (tx!=null) tx.rollback();
       e.printStackTrace();
    } finally {
       session.close();
}
```

Como vemos en el ejemplo, fuera del método tenemos que declarar un nueva instancia de la clase auxiliar *HibernateUtil()*, que inicializaremos dentro de nuestro método. Al llamar al método *getSessionFactory()* lo que haremos es crear un nuevo objeto *SessionFactory*, que asignaremos al objeto *factory*, y de ahí podremos crear la sesión a la base de datos.

3.6. Carga, almacenamiento y modificación de objetos

En este apartado, os explicaremos cómo realizar consultas básicas CRUD con Hibernate.

Las siglas **CRUD** significan *create* (crear), *read* (leer), *update* (actualizar) y *delete* (borrar), que son las tareas principales que se pueden realizar en las bases de datos.

Cuando creamos una aplicación, queremos que sea capaz de realizar cuatro tareas básicas: crear, leer, actualizar y borrar información. A estas funciones se les llama comúnmente funciones CRUD. Una aplicación debe tener la capacidad de realizar estas cuatro funciones, y se ejecutarán siempre en la capa modelo siguiendo la arquitectura MVC.

El paradigma CRUD es común en la construcción de aplicaciones web, ya que proporciona una buena guía para recordar a los desarrolladores cómo construir un modelo de aplicación completo y reutilizable.

Por ejemplo, imaginemos que tenemos una aplicación para hacer seguimiento de los alumnos. En la base de datos tendremos una tabla "alumnos" que contendrá registros, cada fila será un alumno. Para hacer que este sistema de control de alumnos sea útil, queremos asegurarnos de que haya mecanismos claros para completar las operaciones CRUD:

Crear

Esta operación se encarga de insertar un nuevo registro en la base de datos, equivale a un *INSERT* en lenguaje SQL. Tenemos dos opciones: *save()* o *persist()*.

- ***Save()****:* para crear esta función deberíamos crear un método que, al ser llamado, nos cree un nuevo registro de alumno en la base de datos. A este método se le tendrá que pasar el objeto que queramos guardar, en este caso, un objeto

"Alumno" con todos sus atributos llenos de la información a guardar. El método debería encargarse de realizar todas las acciones necesarias para que la información se guarde en la tabla como un nuevo registro. Además, deberá tener un identificador único que permita recuperar los datos.

```
public Integer crearAlumno(String nombre, String apellido, String
dni){
    private static SessionFactory factory;
    util = new HibernateUtil();
      factory = util.getSessionFactory();
      Session session = factory.openSession();

    Transaction tx = null;
    Integer idEstudiante = null;

    try {
       tx = session.beginTransaction();
       Estudiante estudiante = new Estudiante(nombre, apellido,
dni);
       idEstudiante = (Integer) session.save(estudiante);
       tx.commit();
    } catch (HibernateException e) {
       if (tx!=null) tx.rollback();
       e.printStackTrace();
    } finally {
       session.close();
    }
```

Por ejemplo, para guardar una instancia de la clase "Estudiante", deberemos crear un método *crearAlumno()* en la capa *Modelo* de nuestra aplicación. Lo primero que se debe hacer es declarar un *SessionFactory*. A continuación, declararemos una *Session*, que rellenaremos con la llamada *factory.openSession()*. La llamada a este método creará una nueva sesión con la base de datos. A continuación, debemos tener declarada una *Transaction*, que nos va a permitir ejecutar nuestra operación. Asignaremos el valor de la transacción con el método *beginTransaction()*. Después crearemos una nueva instancia de la clase "Estudiante" y le pasaremos todos los datos al constructor para que se cree un objeto con todos los datos.

Para guardar los datos en la base de datos, haremos una llamada al método *save()*. Este método siempre devuelve el identificador del objeto que acaba de guardar en la base de datos, por eso es importante que, al hacer esta llamada, se la asignemos a una variable.

Para que se guarden los datos, deberemos hacer *commit()* de la transacción con el método del mismo nombre. Llamando a este método nos aseguramos de que la operación se realiza y el objeto se persista en la base de datos.

- ***persist()****:* este método está destinado a crear una nueva instancia en la base de datos. Tiene el mismo objetivo que el método *save()*. Al usar el método *persist()*, el estado de eso objeto pasa de *transient* a *persistent*.

Actualización

La operación de actualización *(update)* se encargará de actualizar las columnas de la base de datos con los nuevos datos aportados. Esta operación equivale al *UPDATE* conocido de SQL. En Hibernate tenemos diferentes maneras de actualizar un objeto:

- ***update()****:* es uno de los métodos originales de Hibernate. Se utiliza para actualizar los datos cuando se está seguro de que la sesión no contiene una instancia ya persistente con el mismo identificador.
- ***Merge()****:* es un método nuevo parecido a *update()* que se utiliza si queremos guardar las modificaciones del objeto en cualquier momento sin saber el estado de una sesión.
- ***saveOrUpdate()****:* solo se puede usar en la API de Hibernate. Es similar a *update()*, y se puede utilizar para volver a conectar instancias.

```
public void updateEstudiantes() {
      private static SessionFactory factory;
      util = new HibernateUtil();
      factory = util.getSessionFactory();
      Session session = factory.openSession();

      Transaction tx = null;
      try {
            tx = session.beginTransaction();
            Estudiante  estudiante = (Estudiante)
            session.load(Estudiante.class, new Long(101));
            tx.commit();
            //actualizar valor
            estudiante.setNombre("Juan");
            estudiante.setApellido("Palomo");
            Transaction tx7 = session.beginTransaction();
            session.update(estudiante);

            tx7.commit();
      } catch (HibernateException e) {
            if (tx != null)
                  tx.rollback();
            e.printStackTrace();
      } finally {
            session.close();
      }
}
```

Como vemos en el ejemplo, lo que cambia respecto del resto de ejemplos es la manera de actualizar los datos y los métodos que llamar. En primer lugar, cargamos con el método *load()* el objeto "Estudiante". A continuación, modificamos los valores que nos interesen y llamamos al método *update()* de la sesión pasándole el objeto, seguido de un *commit()* de la transacción. Esta sería la forma más sencilla de actualizar los datos. Si queremos utilizar las otras alternativas, tenemos que sustituir el *update()* por cualquiera de los otros dos métodos.
Debemos tener en cuenta que en este caso también tenemos que englobar las operaciones entre un *try-catch* para capturar el error.

Eliminar

La operación eliminar se encargará de borrar un registro de la base de datos. Es el equivalente a *DELETE* en SQL. El borrado de Hibernate permite borrar un objeto de sesión en estado *transient* siempre que ese objeto contenga al menos el id. Si el objeto está vacío, Hibernate lanzará un error.

```
public void deleteEstudiantes() {
	private static SessionFactory factory;
	util = new HibernateUtil();
	factory = util.getSessionFactory();
	Session session = factory.openSession();

	Transaction tx = null;
	try {
		tx = session.beginTransaction();
		Estudiante  estudiante = new Estudiante();
		Estudiante.setId(1L);
		session.delete(estudiante);
		tx.commit();
	} catch (HibernateException e) {
		if (tx != null)
			tx.rollback();
		e.printStackTrace();
	} finally {
		session.close();
	}
}
```

La manera de implementar este tipo de operación es muy parecida a la utilizada anteriormente. Para eliminar un objeto, debemos crear un método que envuelva todo el procedimiento en un *try-catch* que lance una *HibernateException*. Seteamos el objeto

"Estudiante" con el id. Después, para eliminarlo, solo debemos llamar al método *delete()*. Debemos recordar que, antes de borrar un objeto, tenemos que asegurarnos de que al menos esa instancia tiene un id para identificar el registro de la base de datos que se quiere eliminar.

3.7. Consultas SQL

Esta operación se encarga de recoger la información de un registro de la base de datos. Podemos obtener un listado con todos los resultados o filtrar el contenido de la base de datos pasándole un id a la función de búsqueda. Esta consulta no altera los datos de la base de datos. Equivale a un *SELECT* en SQL, por tanto, es una consulta que no implica cambios en la base de datos.

Antes de ver un ejemplo de cómo realizar esta operación, debemos tener en cuenta diferentes maneras de obtener los datos. La interfaz *Session* tiene diferentes métodos que nos van a permitir obtener los datos de diferente manera según si deseamos solo un resultado o una lista.

Tipo	Método	Descripción
Object	get(Class clase, Serializable id)	Método que devuelve un objeto con el id que le especifiquemos por parámetro.
Object	get(String nombreEntidad, Serializable id)	Método que devuelve un objeto con el id que le especifiquemos por parámetro.
List	createQuery(String consulta)	Método que se encarga de devolver el resultado de la consulta que pasemos por parámetro.
Object	load(Class clase, Serializable id)	Este método devuelve un objeto de la clase y el id que le especifiquemos por parámetro.

Tabla 20. Métodos más destacados para obtener registros en Hibernate.

La implementación de estos métodos sigue una estructura igual que el ejemplo anterior. Primero veremos cómo obtener una lista de estudiantes. Deberemos declarar una sesión y una transacción, como en el ejemplo anterior. A continuación, debemos englobar el siguiente código en un *try-catch* para poder registrar cualquier tipo de error que se pueda producir. Tendremos que empezar la transacción con una llamada a

beginTransaction. Crearemos una lista de estudiantes y la rellenaremos con una llamada al método *createQuery()* de la sesión. Como podemos apreciar, indicamos mediante una *query* qué objeto queremos recuperar de la base de datos. El método *list()* devolverá la lista de los objetos de la consulta ejecutada. Si se produce un error, el *catch* con *HibernateException* capturará el error.

```
public void listaEstudiantes() {
      private static SessionFactory factory;
      util = new HibernateUtil();
      factory = util.getSessionFactory();
      Session session = factory.openSession();
      Transaction tx = null;
      try {
            tx = session.beginTransaction();
            List estudiantes = session.createQuery("FROM
            Estudiante").list();
            for (Iterator iterator = estudiantes.iterator();
            iterator.hasNext();) {
                  Estudiante estudiante = (Estudiante)
                  iterator.next();
                  System.out.print("Nombre: " +
                  estudiante.getNombre());
                  System.out.print("  Apellido: " +
                  estudiante.getApellido());
                  System.out.println("  Dni: " +
                  estudiante.getDni());
            }
            tx.commit();
      } catch (HibernateException e) {
            if (tx != null)
                  tx.rollback();
            e.printStackTrace();
      } finally {
            session.close();
      }
}
```

Para obtener solo un resultado, podemos implementar el resto de los métodos de la tabla. Vamos a ver el ejemplo de uno de esos métodos. El resto la implementación es igual.

```
public void listaEstudiantes() {
	Session session = factory.openSession();
	Transaction tx = null;
	try {
		tx = session.beginTransaction();
		Estudiante estudiante = session.get(Estudiante.class,
		new Integer(2));
		tx.commit();
	} catch (HibernateException e) {
		if (tx != null)
			tx.rollback();
		e.printStackTrace();
	} finally {
		session.close();
	}
}
```

Como podemos ver, el proceso es el mismo. Tenemos que crear los mismos objetos, pero esta vez la sesión hará una llamada al método *get()*. A este método le especificaremos la clase que queremos obtener, en este caso, "Estudiante", y el id del objeto que queremos recuperar. Lo asignaremos a una clase "Estudiante". Si no encuentra un registro, Hibernate lanzará una excepción que capturará el *catch*.

3.7.1. Embebidas

Como hemos visto en el punto anterior, Hibernate nos permite, mediante el método *createQuery()*, crear una consulta personalizada. A través de este método podemos realizar todo tipo de consultas más complejas, lo que nos aportará mucha más funcionalidad. Es aquí donde entra el lenguaje propio de Hibernate: el HQL.

El HQL (Hibernate Query Language) es un lenguaje propio de Hibernate orientado a objetos para hacer consultas a base de datos con un lenguaje propio.

Estas son algunas de las características más importantes que nos proporciona HQL:

- **Soporte operaciones relacionales**: HQL permite representar consultas SQL en forma de objetos. HQL usa clases y atributos o propiedades en lugar de tablas y columnas.
- **Resultado en forma de objetos**: las sentencias de consulta de datos pueden retornar un objeto o un listado de objetos. Se elimina la necesidad de crear un objeto de cero con los valores del resultado de la sentencia, pues HQL se encarga de realizar esta acción.
- **Facilidad de aprendizaje**: el lenguaje HQL, al ser muy similar a SQL, es muy fácil de aprender.
- **Consultas polimórficas**: Hibernate se encarga de cuadrar el tipo de objeto que declaremos con el obtenido de la consulta. Se pueden mezclar los tipos de objetos sin que se produzca un error.
- **Características avanzadas**: en HQL se introducen nuevas funciones, como la paginación, y nuevas cláusulas, como *fetch join*, *inner* y *outer joins*. También soporta proyecciones, funciones *max* o *avg*, *order by*, *group by*, y subconsultas.
- **Independiente**: las consultas que realicemos con HQL son independientes del gestor de BBDD. HQL es solo para Hibernate, no se puede utilizar en la aplicación de nuestra BBDD.

Cláusula FROM

Es la forma más sencilla de realizar una consulta en base de datos. A diferencia de SQL, no es necesario introducir la sentencia *SELECT*. Se estructura de esta manera:

```
FROM org.ilerna.ejemplos.Objeto
FROM Objeto
```

Para hacer una selección sencilla, tenemos la opción de añadir solo el nombre del objeto, que es lo más práctico, o añadir toda la ruta de paquetes donde se encuentra el objeto que seleccionar, como vemos en el primer ejemplo. Lo que hará este tipo de consulta es devolver todos los registros de la base de datos que correspondan a ese objeto.

Dentro de la cláusula FROM, podemos añadir más de una clase, separadas mediante una coma. El resultado mostrará todos los registros de las dos tablas.

```
FROM Estudiante, Profesor
```

Cuando tenemos más de un objeto que seleccionar, es recomendable añadirle un alias para poder tener identificado cada objeto. HQL también contempla la cláusula *WHERE*, que nos va a permitir afinar las búsquedas.

El alias ***e*** hará referencia al objeto "Estudiante" y seleccionará todos los registros que tengan como e.id= 1.

```
FROM Estudiante e WHERE e.id=1
```

Asociaciones y *joins*

Cuando queremos realizar vínculos entre un objeto y otro, debemos utilizar *joins*. HQL incluye cuatro tipos distintos: *inner join*, *left outer join*, *right join* y *full join*. Esta cláusula nos va a permitir realizar consultas a múltiples objetos y obtener resultados de múltiples objetos estableciendo relaciones entre ellos.

Para ello, deberemos añadir alias a cada uno de los objetos asociados y establecer la relación con el otro objeto con *inner join*. Un alias será como un mote o una abreviación del objeto para hacer referencia a él, lo que nos será útil para añadir condiciones a la cláusula *WHERE*. Para añadir una relación con alias, los dos objetos deben estar relacionados mediante un campo. Lo haremos de esta manera:

```
FROM Estudiante e inner join e.profesor p WHERE e.id=1
```

Como vemos, tenemos un objeto "Estudiante" con alias *e*. Este objeto debe tener declarado en su entidad un objeto "Profesor" para poder establecer la relación con la entidad "Profesor". Si es así, se puede realizar un *inner join* desde el alias *e.* añadiendo el nombre del atributo en esa clase, en este caso, "Profesor".

El resto de *joins* funcionan de la misma manera, se tiene que establecer la relación mediante los atributos de cada entidad.

3.7.2. Gestión de transacciones

Es la parte más importante que gestiona Hibernate. Podemos considerar una transacción una manera de representar una acción que se realizará en la base de datos.

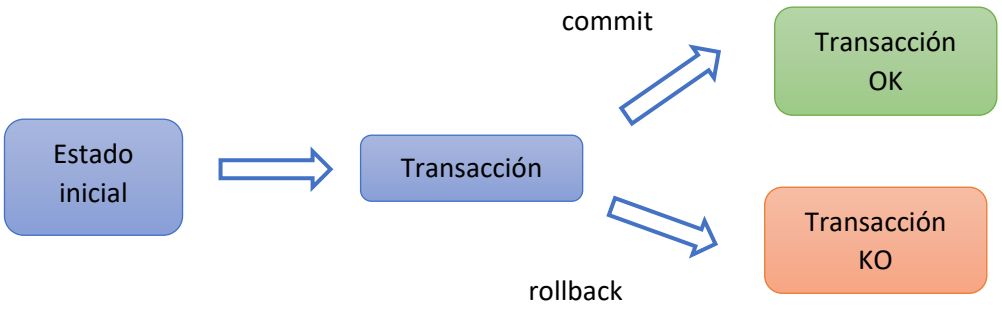


Ilustración 41. Estados de una transacción.

En Hibernate tenemos la interfaz *Transaction*, que hemos visto en apartados anteriores y que mantiene la abstracción de la implementación de la transacción. Una transacción está asociada a una sesión y se instancia mediante *session.beginTransaction()*. Para poder dominar la gestión de transacciones, debemos memorizar estos métodos y sus funciones, ya que nos serán de gran utilidad para trabajar con Hibernate:

Tipo	Método	Descripción
void	begin()	Método que se encarga de iniciar una transacción.
void	commit()	Método que ejecuta la transacción.
void	rollback()	Se encarga de tirar atrás todas las acciones que se han realizado hasta el momento.
void	setTimeout(int segundos)	Este método establece un *timeout* a la transacción. Si durante el tiempo establecido no se ha podido ejecutar la acción, se lanza un *timeout*.
boolean	isAlive()	Verifica si la transacción aún está activa.
boolean	wasCommited()	Verifica si la transacción se ha comiteado correctamente.

boolean	wasRolledBack()	Verifica si la transacción ha realizado *rollback* correctamente.

Tabla 21. Métodos más importantes de la interfaz Transaction.

```
Session session = null;
Transaction tx = null;

try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();
    //some action

    tx.commit();
}catch (HibernateException ex) {
    ex.printStackTrace();
    tx.rollback();
}
finally {
      session.close();
}
```

Esta es una estructura básica para gestionar transacciones en Hibernate. La transacción se debe declarar con el objeto *Transaction*, que se rellenará con la llamada del método *beginTransaction()* de la interfaz *Session*. Siempre que gestionemos con transacciones, deberemos realizar una llamada al método *commit()* para que se ejecuten todas las operaciones que realicemos. Si se produce algún tipo de error, se recogerá en el *catch*. En este caso, es muy recomendable realizar un *rollback()* para deshacer todas las acciones realizadas hasta el momento.

3.7.3. Prueba y documentación de las aplicaciones desarrolladas

Para poner en práctica todo lo aprendido, vamos a hacer un test de las operaciones CRUD para verificar que funcionan correctamente. Para ello, será necesario tener creado un proyecto Java o un proyecto Maven.

En primer lugar, tenemos que tener creada la base de datos y la tabla de la entidad. Si hemos seguido los diferentes ejemplos de los temas, tendríamos que tener un proyecto con una entidad "Estudiante" y una entidad "Profesor". Usaremos estos objetos para realizar las pruebas con Hibernate. Tendremos que haber creado la clase *HibernateUtil* para realizar la conexión a la base de datos. En el apartado 3.3 se ha visto cómo implementar esta clase.

Para realizar las pruebas, tenemos que tener en nuestro proyecto esta estructura:

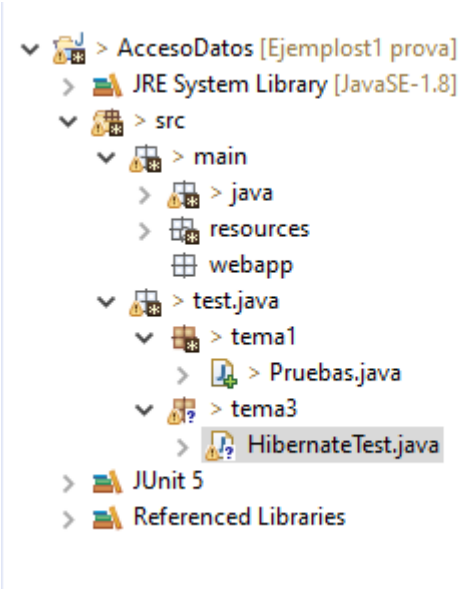


Ilustración 42. Estructura del proyecto Java.

La prueba que realizaremos en primer lugar será comprobar si Hibernate es capaz de seleccionar datos de la tabla "Estudiante".

Dentro de la carpeta src, crearemos un paquete test donde añadiremos todas las clases test que usemos para realizar las pruebas. El objetivo principal de estas pruebas es verificar que todo el código implementado funciona correctamente, y nos ayudará a prevenir futuros fallos en el código.

En primer lugar, tenemos que verificar que tenemos mapeada la entidad que queremos testear, ya sea con el fichero .hbm.xml o con las anotaciones. Estas son las dos opciones, solo es necesario elegir una de las dos:

```
@Table(name = "estudiante")
public class Estudiante {
	@Id
	@GeneratedValue
	@Column(name = "id")
	private int id;

	@Column(name = "dni")
	@OrderBy("desc")
	private String dni;

	@Column(name = "nombre")
	private String nombre;

	@Column(name = "apellido")
	private String apellido;

	@Column(name = "edad")
	private int edad;


	//GETTERS Y SETTERS

}
```

Para hacer las pruebas, tenemos que tener una clase DAO creada con todos los métodos con las operaciones CRUD. Una clase DAO es el tipo de clase que se usa para realizar operaciones en base de datos siguiendo la metodología MVC, por ello, es recomendable que tengamos diferentes capas de gestión. Cada entidad debería tener su clase DAO propia. En esta ocasión, tendremos la clase "EstudianteDaoImpl", que implementará todas las operaciones CRUD. Debemos tener algo parecido a esto:

```
public class EstudianteDaoImpl {

	private static SessionFactory factory;

	public Integer crearAlumnoObjeto(Estudiante estudiante) {
		private static SessionFactory factory;
		util = new HibernateUtil();
		factory = util.getSessionFactory();
		Transaction tx = null;
		Integer idEstudiante = null;

		try {
			tx = session.beginTransaction();

			idEstudiante = (Integer) session.save(estudiante);
			tx.commit();
		} catch (HibernateException e) {
			if (tx != null)
				tx.rollback();
			e.printStackTrace();
		} finally {
			session.close();
		}
		return idEstudiante;
	}
	//Continuación para el resto de métodos.
}
```

Seguidamente, tenemos que crear la clase que realizará los test. Esta clase la tendremos que situar en otro paquete distinto. Crearemos dentro de src un paquete test y otro Java que contendrá las diferentes clases "Test" que creemos. Es importante tener una estructura y orden en el proyecto para saber dónde se encuentran cada una de las clases y para qué se utilizan.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
<hibernate-mapping>
      <class name="main.java.objetos.Estudiante"
table="estudiante">
      <id name="id" type="integer">
            <generator class="assigned"></generator>
      </id>
      <property name="dni" column="dni" type="string" />
      <property name="nombre" column="nombre" type="string" />
      <property name="apellido" column="apellido" type="string"
/>
      <property name="edad" column="edad" type="integer" />
</class>
</hibernate-mapping>
```

Antes de empezar, detallaremos las etiquetas y métodos más importantes que usaremos para construir nuestros test de pruebas.

TAG	DESCRIPCIÓN
@Test	Identifica un método como un método test.
@Before	Indica qué se ejecuta antes de cada test. Se usa para preparar el entorno del test antes de ejecutarlo.
@After	Indica qué se ejecutará después de cada test. Se usa para limpiar el entorno después de la ejecución de cada test.
@BeforeClass	Se ejecuta solo una vez, antes de que se ejecuten todos los test. Se usa, por ejemplo, para conectar con la base de datos.
@AfterClass	Se ejecuta solo una vez después de ejecutar todos los test. Se usa para limpiar el entorno.
@Ignore or @Ignore ("why disabled")	Marca ese test como deshabilitado.
@Rule	Establece una regla, añade funcionalidades extra a nuestro test.

Tabla 22. Relación de etiquetas más usadas para los test con JUnit.

METODO	DESCRIPCIÓN
fail([message])	Se usa para hacer que el método falle y descubrir qué partes del código no se contemplan antes de desarrollar. El parámetro *message* es opcional.
assertTrue([message,]boolean condition)	Comprueba si la condición es *true*.
assertFalse([message,]boolean condition)	Comprueba si la condición es *false*.
assertFalse([message,]boolean condition)	Hace test de dos valores para comprobar si son el mismo.
assertEquals([message,] expected,actual, tolerance)	Comprueba si un *float* o un *double* son iguales.
assertNull([message,] object)	Comprueba si un objeto es *null*.
assertNotNull([message,] object)	Comprueba si el objeto no es *null*.
assertSame([message,] expected, actual)	Comprueba si las dos variables hacen referencia al mismo objeto.
assertNotSame([message,] expected, actual)	Comprueba si las dos variables no hacen referencia al mismo objeto.

Tabla 23. Relación de métodos más importantes para la ejecución de JUnit.

La clase que crearemos se llamará "TestEstudiante". Todos los test relacionados con la entidad "Estudiante" se crearán en esta clase. Para empezar, cada método de nuestro test deberá estar anotado con la anotación *@Test*.

```
@Test
public void testBorrado() {
	boolean b = true;
	EstudianteDaoImpl dao = new EstudianteDaoImpl();
	Estudiante estudiante = new Estudiante();
	estudiante.setNombre("Juan");
	estudiante.setApellido("Palomo");
	estudiante.setDni("112222444H");
	estudiante.setEdad(21);
	estudiante.setId(1);

	assertTrue(dao.deleteEstudiantes(estudiante));

}
```

Lo esencial de este tipo de métodos test es corroborar que realizan la función para la cual han sido creados. En este ejemplo, realizaremos un test de borrado. Primero tenemos que hacer una nueva instancia de la clase DAO, porque usaremos el método de borrado para comprobar si funciona correctamente. La clase DAO se usará porque es la que implementa esta acción contra el objeto y es la que se encargará de realizar la conexión con la base de datos.

A continuación, crearemos un nuevo objeto "Estudiante", le asignaremos los valores y, con el método *assertTrue()*, comprobaremos si, haciendo la llamada a *deleteEstudiantes()*, se realiza el borrado correctamente. Si el test no falla, el *assertTrue()* dará *true*, ya que el resultado de la prueba será positivo; de lo contrario, será *false*. Para este test, podemos usar otro tipo de método, como *assertTrue()* o *assertFalse()*, para hacer las comprobaciones.

Vamos a ver otro ejemplo: en este caso, haremos un test del método *updateEstudiantes()*, que se encarga de hacer un *update* con los nuevos datos de esa entidad.

```
public void updateEstudianteTest() {
	boolean b=true;
	EstudianteDaoImpl dao = new EstudianteDaoImpl();
	Estudiante estudiante = new Estudiante();
	estudiante.setNombre("Juan");
	estudiante.setApellido("Palomo");
	estudiante.setDni("112222444H");
	estudiante.setEdad(21);
	estudiante.setId(1);

	assertTrue(dao.updateEstudiantes(estudiante));
}
```

El procedimiento es el mismo, pero, en esta ocasión, el DAO hace la llamada al método *updateEstudiantes()*.

Tendremos que realizar el test de todos los métodos que contenga el DAO para verificar que todos funcionan correctamente.

4. Bases de datos objeto-relacionales y orientadas a objetos

En este tema 4 nos centraremos en las bases de datos objeto-relacionales y orientadas a objetos. Para empezar:

Una **base de datos objeto-relacional** (BDOR) es un tipo de BBDD que se caracteriza por haber evolucionado del modelo relacional a la orientación a objetos, convirtiéndose en una base de datos híbrida, ya que usa las dos tecnologías.

Este tipo de base de datos es uno de los más utilizados, dado que su modelo permite representar problemas reales y administrar los datos de manera dinámica.

Por otro lado, tenemos otro tipo de base de datos: la base de datos orientada a objetos.

Una **base de datos orientada a objetos** se caracteriza por representar la información que se guarda mediante objetos, tal y como se realiza en el lenguaje de programación orientado a objetos.

4.1. Características de las bases de datos objeto-relacionales

La principal característica de una base de datos objeto-relacional es su capacidad de poder crear objetos propios para gestionar un tipo de datos personalizado. Es decir, nosotros podremos crear nuestro tipo de objetos para usarlo en nuestra base de datos. Nos ofrecerán una complejidad interna a diferentes niveles.

Por ejemplo, podremos especificar un objeto como un tipo de dato y el valor será una instancia de este tipo. Un objeto, por tanto, será una instancia de este tipo de objeto.

Tipo de objeto Atributos	Instancia Objeto 1	Instancia Objeto 2
• id • nombre • apellido • mail	• id: 1 • nombre: Laura • apellido: Lopez • mail: mail@mail.com	• id: 2 • nombre: Juan • apellido: Martinez • mail: info@mail.com

Ilustración 43. Ejemplo instancia de objetos.

Este tipo de base de datos, al estar basado en el modelo objeto-relación, nos va a permitir usar los beneficios de la herencia entre objetos.

Se conoce como **herencia** a la relación entre un tipo de objeto más general y otro más específico.

Al establecer esta relación, un tipo objeto se deriva del otro extendiendo su funcionalidad, siendo el mecanismo fundamental para implementar el polimorfismo y la reutilización.

El **polimorfismo** es la capacidad que tiene un objeto de ofrecer una respuesta diferente e independiente en función de los parámetros que se utilizan cuando se invoca el objeto.

```
class Animal {
	public void come() {
		System.out.println("Pienso");
	}
}
class Leon extends Animal {
	public void come() {
		System.out.println("Carne");
	}
}
class Foca extends Animal {
	public void come() {
		System.out.println("Pescado");
	}
}
```

Con este ejemplo podemos entenderlo mejor. Tenemos dos clases diferentes: la clase "Foca" y la clase "León", que heredan de la clase "Animal". Esta clase tiene un método abstracto *come()* que se implementa de forma distinta en las subclases. Cuando se crea un nuevo objeto "León", al llamar al método *come()* se mostrará un mensaje diferente que si una nueva instancia de "Foca" llama al método *come()*.

Por tanto, el **polimorfismo** lo que hará es sobrescribir métodos a la hora de implementar un tipo de objeto usando la herencia.

Otra característica importante de este tipo de base de datos es la ocultación de datos de un objeto de modo que solo se permita su modificación a partir de las operaciones definidas por este objeto. Este concepto se conoce como **encapsulación**. Este aislamiento protege los datos que contiene cada objeto, obstaculizando la modificación o la eliminación por parte de un usuario externo.

Este tipo de base de datos es muy fácil de adaptar en diferentes aplicaciones. Goza de una gran adaptabilidad porque es compatible con otras bases de datos de tipo relacional, sin necesidad de reescribir la base de datos.

4.2. Gestión de objetos con SQL. Especificaciones en estándares SQL; ANSI SQL 1999; nuevas características orientadas a objetos

El lenguaje SQL es el lenguaje característico para las operaciones con una base de datos objeto-relacional. Fue creado por IBM en 1989 y desde entonces se han ido realizando revisiones del lenguaje. En su primera versión, se definieron los siguientes puntos:

- El lenguaje de definición de datos (LDD): se definieron las instrucciones *create*, *alter* y *drop*, que sirven para crear, definir o borrar bases de datos.
- El lenguaje de manipulación de datos (LMD): establecieron las instrucciones de gestión de tablas: *select*, *insert*, *delete*, *update*, *commit* y *rollback*.
- El lenguaje de control de datos (LCD): establecieron todas las operaciones para otorgar o revocar permisos: *grant* y *revoke*.

En esta primera definición se establecieron las operaciones más básicas y esenciales, pero, a medida que se iba utilizando más el lenguaje SQL, se creaban nuevas necesidades para definir más el funcionamiento, por lo que en 1992 se desarrolló otra parte del

estándar. Esta fue una de las revisiones del estándar más importantes, ya que se añadieron muchos cambios. En resumen, se añadieron estas características:

- Nuevos tipos de datos, como *DATE*, *TIMESTAMP*, *TIME* e *INTERVAL*, y diferentes tipos de *string*, como *BIT*, *VARCHAR* y *NATIONAL CHARACTER*.
- Cursores.
- Posibilidad de definir esquemas.
- Creación de tablas temporales.
- Transacciones.
- Operaciones: *union join*, *natural join*.
- Nuevas operaciones para la definición de datos: ALTER y DROP.
- Posibilidad de controlar los privilegios de los usuarios.
- Se aseguró que todos los estándares creados en el pasado o los que se crearan en el futuro fueran compatibles los unos con los otros.

En esta ocasión, os explicaremos con más detalle la revisión de 1999, qué incorpora y qué afectación tuvo para las bases de datos objeto-relacionales.

En 1999 se realiza la cuarta revisión del lenguaje SQL. Se introdujeron nuevas funcionalidades, como cambios en los tipos de datos, nuevas expresiones de tablas y consultas recursivas, y nuevos aspectos relacionales. Los cambios realizados podemos separarlos en nuevas funcionalidades relacionales y nuevas funcionalidades orientadas a objetos:

4.2.1 Funcionalidades relacionales

El lenguaje SQL vio la necesidad de realizar ciertos cambios y añadir funcionalidades a este lenguaje para mejorar las capacidades relacionales de este lenguaje para modelar los datos.

1. **Nuevos tipos de datos**:

   En esta revisión del estándar SQL se vio la necesidad de añadir cuatro nuevos tipos de datos que dieran más funcionalidad al lenguaje SQL.

   - **Nuevo tipo de dato *large object (LOB)* y sus variantes: *CLOB* y *BLOB***. Existen dos tipos de datos *LOB*. El primero, *CLOB (character large objects)*, se caracteriza por ser un dato que almacena largas cadenas de caracteres, y el *BLOB* es un tipo de dato binario de gran tamaño.
   - ***Boolean**:* añade estos tipos de datos que antes se definían como *BIT*.

- ***Array:*** este tipo de dato permite guardar una colección de valores directamente en una columna de la tabla. Cuando creemos una nueva columna, tendremos que asignar el tipo de datos que contendrá el *array* y añadir ARRAY con el tamaño correspondiente, tal y como vemos en el ejemplo.

```
DIAS VARCHAR(10) ARRAY(8)
```

"DIAS" corresponderá al nombre de la columna; VARCHAR(10) significa que el *array* contendrá *strings* de máximo 10 caracteres; y ARRAY(8), que esa columna será un *array* de *strings* de máximo 8 valores.

- **Colecciones**: se añaden las colecciones ***ROW***. Este tipo permite almacenar de forma directa colecciones enteras de datos, tanto de tipo básico como estructurado.
- Relaciones entre tipos distintos.

2. **Nuevos predicados**:

Un **predicado** es una expresión básica de SQL que se incluye entre la cláusula y el nombre del campo que queremos consultar.

En esta revisión se añaden dos predicados, DISTINCT y LIKE. El primero se encarga de eliminar los repetidos de una consulta SQL. Cuando tenemos una consulta larga, que une más de una tabla, en más de una ocasión puede generar duplicados de los registros, por eso se utiliza DISTINCT para evitar este duplicado, o simplemente en una consulta simple se añade para evitar duplicados.

Por otro lado, tenemos LIKE. Este predicado es un operador lógico que se utiliza

```
select distinct nombre, apellido, dni from estudiante where
apellido= ‘Garcia’;
```

después del WHERE para especificar un patrón en una columna. Nos será útil para comparar si el valor que establecemos después del LIKE se encuentra en las columnas que especifiquemos de una tabla.

```
select * from estudiante where apellido LIKE ‘Garcia’;
```

En este caso, buscará en la columna "apellido" si coinciden registros con el apellido "García".

3. Cambio en la semántica SQL:1999.
4. **Más seguridad**: se añaden permisos por roles y los roles que tengan permiso también pueden añadir permisos.

5. *TRIGGERS:* se añaden los disparadores o *triggers*.

> Los *triggers* son objetos de la base de datos que ejecutan acciones al producirse un evento.

Actualmente, la última y octava versión del **estándar es la de 2016**. Es la más reciente que existe en la actualidad. Vistas las necesidades cambiantes de las bases de datos, en esta versión se añaden más de 44 nuevas funcionalidades opcionales, detallamos las más importantes:

- Posibilidad de crear JSON: se añaden funciones que dan la posibilidad de crear, acceder o verificar los datos de un JSON.
- Reconocimiento de patrones en una fila: se añade la posibilidad de establecer un patrón en las filas siguiendo una expresión regular.
- Se añade el formato y el análisis de las fechas y la hora.
- Nueva función *LISTAGG:* esta función se encarga de transformar los resultados de una operación SQL en un grupo de columnas dentro de un *string*.
- Nuevo tipo de dato *DECFLOAT:* devuelve un *float* de un valor con un tipo de dato distinto.

4.3. Acceso a las funciones del gestor desde el lenguaje de programación

En una aplicación realizada con Java, es posible realizar una conexión con diferentes bases de datos. Para poder utilizar el lenguaje PL/SQL es necesario la utilización de otro tipo de base de datos más sofisticada que la que utilizamos en el tema 1. En esta ocasión, elegiremos una base de datos Oracle, que nos va a permitir trabajar con PL/SQL.

Las ventajas de trabajar en Oracle son:

- Está basado en el modelo relacional.
- Soporta PL/SQL: soporta este tipo de lenguaje mucho más dinámico, como también SQL.
- Control de acceso a los datos: Oracle utiliza tecnología muy avanzada para tener control de acceso de los datos por parte de los usuarios.
- Alta protección de datos: mantiene una alta seguridad para los datos en aplicaciones a producción, como también mantiene una buena gestión de copias de seguridad de los datos.

Instalación del entorno y *drivers*

Para poder trabajar con Oracle, debemos tener instalado en nuestro ordenador la versión de la base de datos de Oracle. Descargaremos el programa desde este enlace:

https://www.oracle.com/database/technologies/xe-downloads.html

Para realizar la instalación de Oracle, podemos seguir este tutorial, donde nos indicarán paso a paso cómo podemos realizarlo. Sobre todo, es importante memorizar o anotar en algún sitio el nombre de la base de datos que vamos a crear y la contraseña, ya que será la que utilizaremos para realizar las prácticas.

https://www.oracletutorial.com/getting-started/install-oracle/

Una vez instalada la base de datos Oracle, tendremos que instalarnos el SQL Developer, que es la aplicación que nos va a permitir gestionar las conexiones con nuestra base de datos Oracle. Podemos realizar la instalación accediendo a este enlace:

https://www.oracle.com/database/technologies/appdev/sql-developer.html

Una vez instalado, tendremos que abrir este programa y veremos que, en la parte superior izquierda, tenemos la opción de crear una nueva conexión a la base de datos:

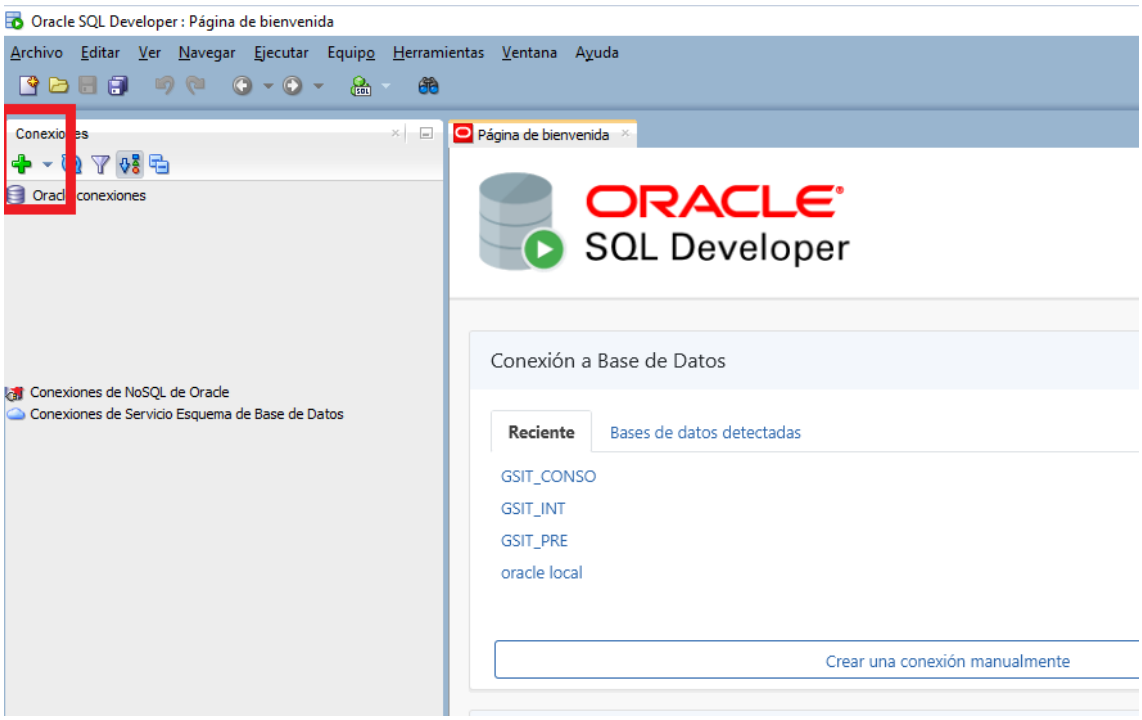
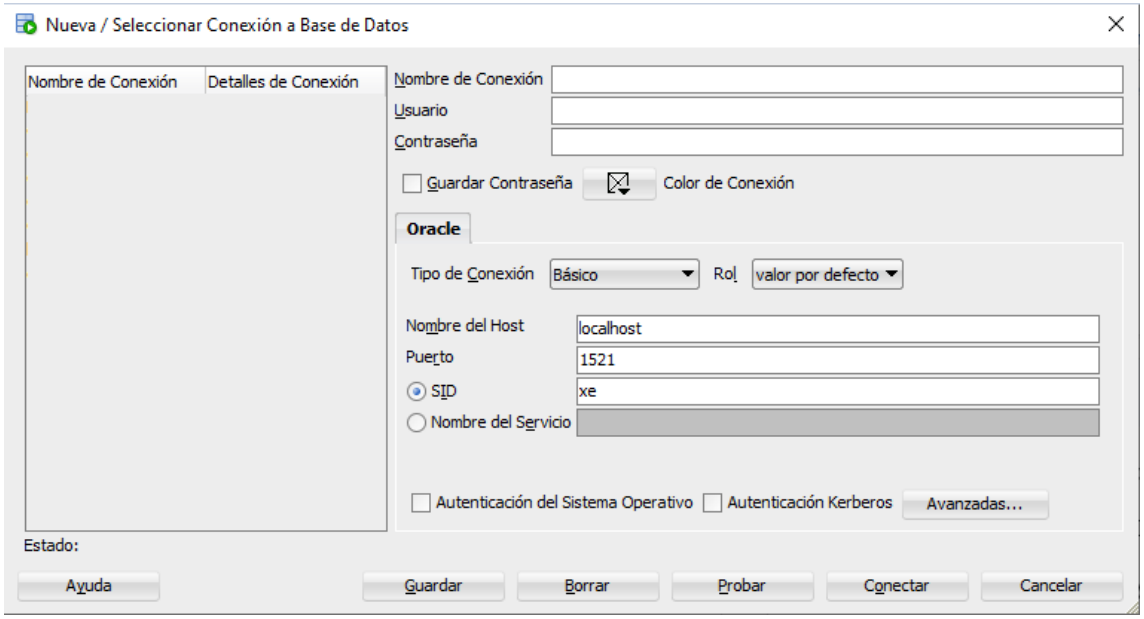


Ilustración 44. Captura para mostrar cómo se realiza una nueva conexión.

Se nos abrirá una ventana como esta:



En esta pantalla tendremos que configurar los datos de conexión que hemos rellenado durante la instalación. Con este paso, lo que vamos a hacer es crear una conexión con el usuario del sistema de esta base de datos, lo que nos va a permitir gestionar todo lo que realicemos en Oracle. Es recomendable tener un usuario *admin* como el que hemos creado durante la instalación para poder crear los usuarios y las distintas bases de datos de nuestras aplicaciones.

En este caso, en *Nombre de Conexión* escribiremos un nombre para identificar la conexión. En *Usuario* pondremos el que hemos creado durante la instalación y su contraseña. Es por este motivo que es importante guardar esta información, ya que es esencial para poder administrar las conexiones.

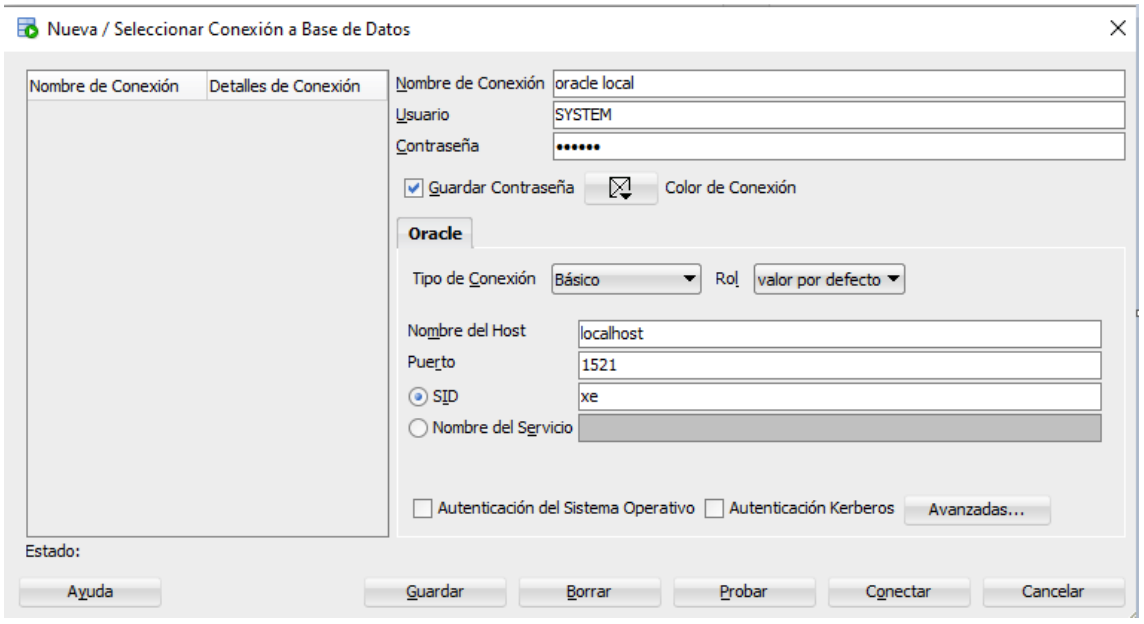


Ilustración 45. Datos que usaremos para la conexión.

Para *Nombre del Host* escribiremos "localhost", ya que trabajaremos en un entorno local. El puerto es el 1521 por defecto y el SID es el "xe", que también aparece por defecto durante la instalación. Antes de guardar la conexión, es recomendable darle al botón de probar para comprobar que la conexión se realiza correctamente. Si no aparece como correcto, tendremos que verificar los datos que hemos introducido. Tendremos que tener algo así al finalizar:

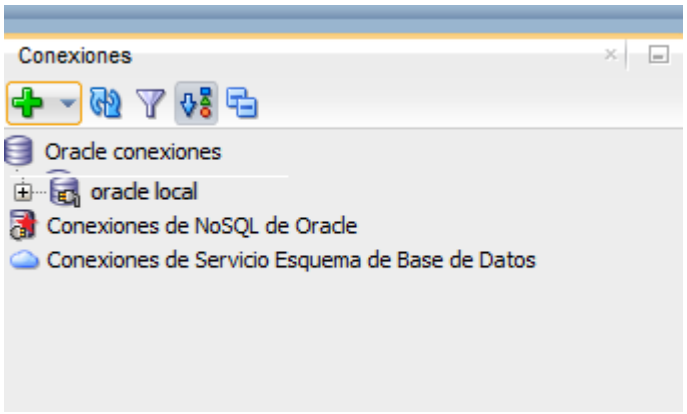


Ilustración 46. Conexión a la base de datos de oracle realizada correctamente.

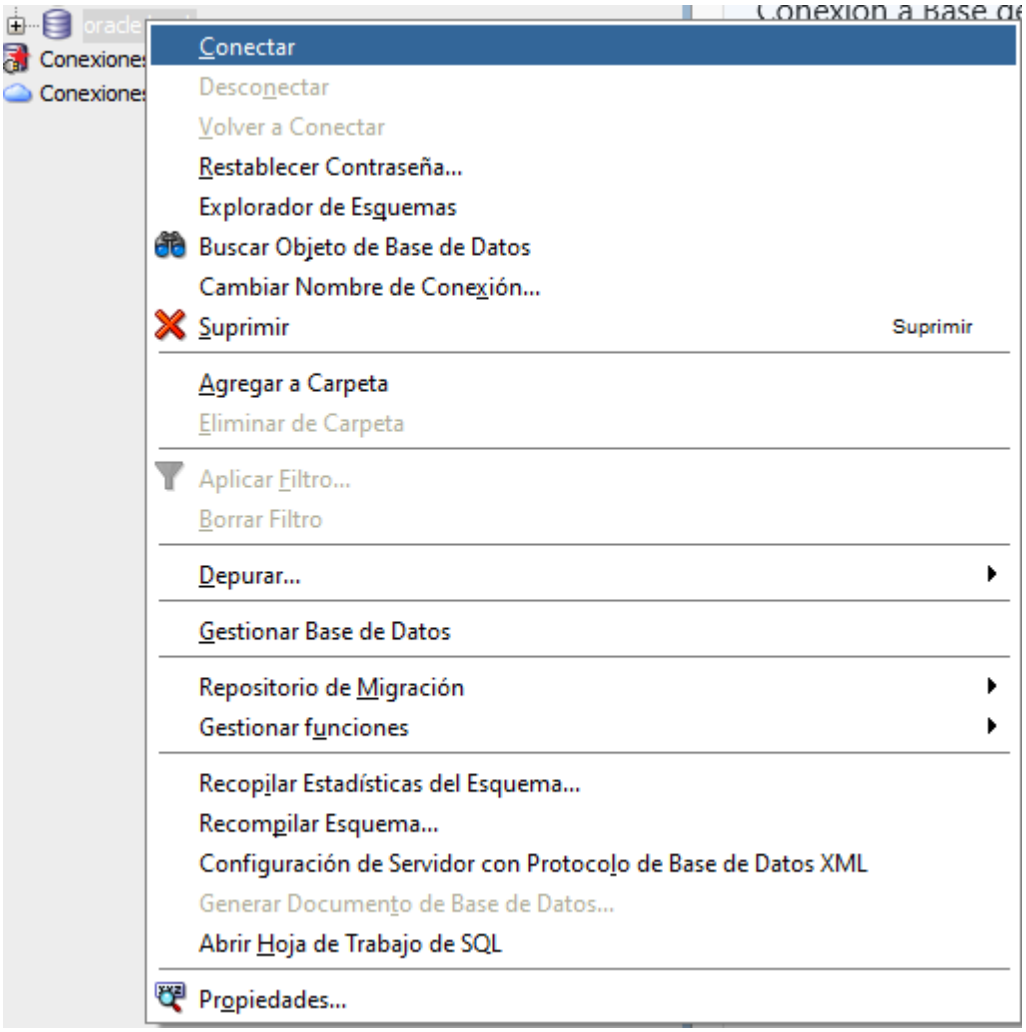


Ilustración 47. Proceso para conectarse a la base de datos.

Para conectarnos a la base de datos, bastará con pulsar con el botón derecho del ratón encima de la conexión y conectar.

A continuación, deberemos crear un usuario para nuestra aplicación usando esta sentencia:

```
create user ilerna identified by 'contrasena';
```

Podemos usar cualquier nombre y cualquier contraseña, y debemos guardarla bien, porque nos será de utilidad para establecer más adelante la conexión entre la base de datos y nuestra aplicación. Tendremos que ejecutarlo en la conexión que hemos abierto a nuestra conexión *admin* de la base de datos.

Seguidamente crearemos una nueva base de datos para nuestra aplicación y le daremos a nuestro usuario permisos:

```
create database tema4;
GRANT ALL PRIVILEGES TO ilerna;
```

Con esto ya tenemos la base de datos configurada. Lo más recomendable es crear una nueva conexión, como hemos hecho en los pasos anteriores, con los datos de esta conexión nueva, lo que nos va a permitir ver las tablas de nuestra base de datos.

Una vez instalada la base de datos, tenemos que añadir el *driver* JDBC de Oracle (OJDBC) a proyecto Java. Como nosotros usamos la versión de Java 8, necesitamos descargar la versión 8 del *driver* de Oracle. Si usáramos una versión Java 10 o superior, necesitaríamos la versión de OJDBC 10.

Podemos descargárnoslo desde este enlace:

https://download.oracle.com/otn-pub/otn_software/jdbc/ojdbc8.jar

Ya descargado, deberemos añadir la librería a nuestro proyecto Java. Tenemos dos métodos para realizar esta tarea según qué tipo de proyecto tengamos. Si tenemos un proyecto Java, tendremos que dirigirnos con el botón derecho a nuestro proyecto e ir a *Properties > Java Built Path*. Cuando se nos abra esta pestaña, deberemos darle al botón *Add External JARs* y añadir el .jar descargado.

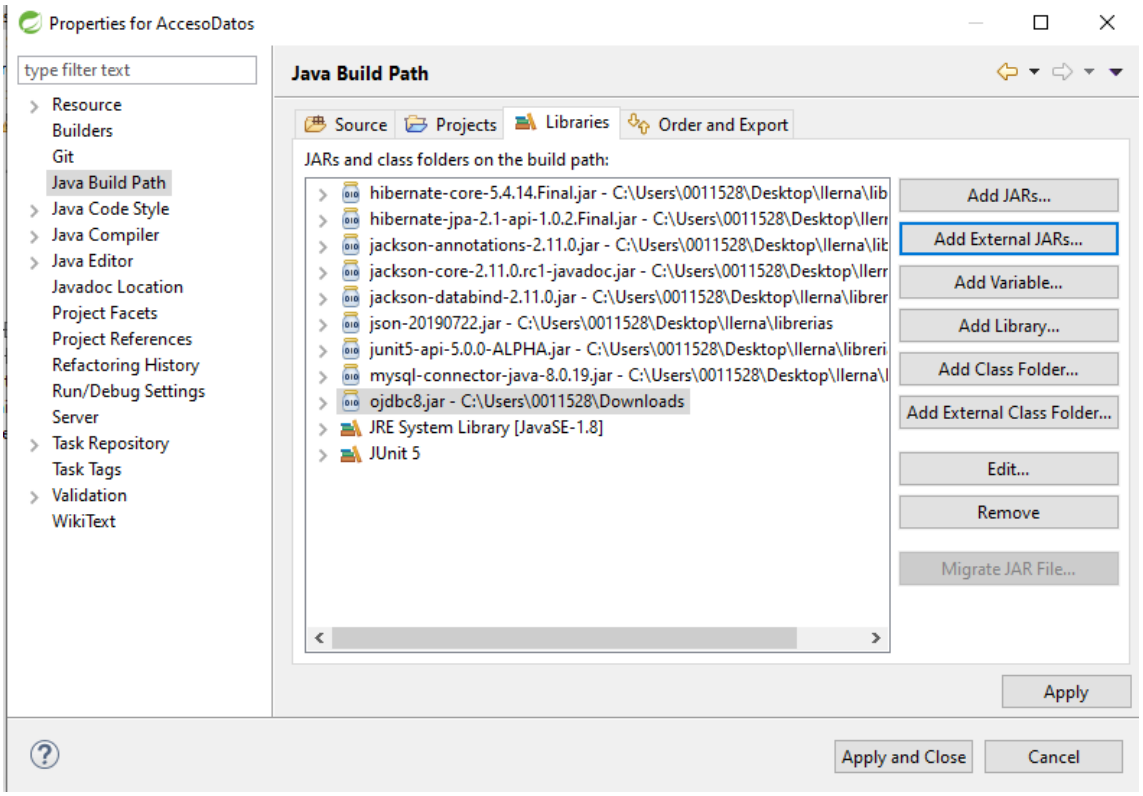


Ilustración 48. Ventana que nos permitirá añadir el driver de Oracle a nuestra aplicación.

Por otro lado, si tenemos un proyecto Maven, el proceso será distinto. Debido a las restricciones de licencia de Oracle, el driver JDBC de Oracle no está disponible en el repositorio público de Maven. Para utilizar el controlador Oracle JDBC con Maven, debes descargarlo e instalarlo en su repositorio local de Maven manualmente. Para ello, deberemos añadir a nuestro pom.xml esta dependencia:

En *systemPath*, tenemos que añadir la ruta donde nos hemos descargado el .jar de OJDBC.

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc8</artifactId>
    <version>1.0</version>
    <scope>system</scope>
    <systemPath>{nombre_ruta}/ojdbc8.jar</systemPath>
</dependency>
```

Preparación de la conexión con la base de datos

A continuación, tenemos que preparar la URL que utilizaremos para realizar la conexión a la base de datos. La sintaxis de la URL para la base de datos de Oracle sigue esta estructura:

```
jdbc:oracle:<tipodriver>:@<basededatos>
o
jdbc:oracle:<tipodriver>:<nombreusuario>/<contraseña>@<basededatos>
```

El tipo de *driver* puede ser: *thin*, *oci* o *kprb*.

Tabla 24. Tipos distintos del driver de Oracle.

Tipo	Descripción	Tipo driver
Driver thin	Para aplicaciones enfocadas a cliente sin una instalación Oracle.	*thin*
Driver OCI	Para aplicaciones enfocadas a cliente con una instalación Oracle.	*oci*
Driver thin servidor	Es igual que el *driver thin*, pero se ejecuta dentro de un servidor de Oracle con acceso remoto.	*thin*
Driver interno servidor	Se ejecuta dentro del servidor elegido.	*kprb*

Para nuestra aplicación Java, utilizaremos el tipo de *driver thin*, ya que tenemos una instalación Oracle y para que funcione la aplicación debemos arrancarla en un servidor.

Para el *nombreusuario*, debemos utilizar el usuario y la contraseña de acceso para la base de datos que hemos creado en pasos anteriores.

Con los datos de la base de datos creada en pasos anteriores y el usuario y la contraseña, podemos acabar de construir la URL que necesitaremos para realizar la conexión a la base de datos. Tendremos que indicar el nombre del *host*, en este caso, "localhost", el puerto 1521 y el nombre de la base de datos, todo separado por dos puntos. Tiene que quedar algo parecido a esto:

```
jdbc:oracle:thin:ilerna/contrasena@localhost:1521:tema4
```

Registrar el *driver* de Oracle JDBC

El proceso para registrar el *driver* de Oracle es muy parecido al que ya vimos durante el tema 1, pero en esta ocasión el nombre de la clase que vamos a utilizar será este:

```
oracle.jdbc.OracleDriver
```

Para realizar el registro del *driver* se utiliza:

```
Class.forName("oracle.jdbc.OracleDriver");
```

Des de la versión 6 de Java, registrar el *driver* como hemos explicado se convierte en un paso opcional. Siempre que añadamos la librería de OJDBC a nuestro proyecto, el registro del *driver* se hará de manera automática. No está de más saber cómo se realiza de manera manual, ya que en según qué tipo de base de datos que gestionemos puede sernos útil.

Establecer la conexión con Oracle

Con OJDBC el proceso de conexión con la base de datos es igual que en JDBC. Vamos recordar cómo se realizaba la conexión.

Para establecer la conexión con la base de datos, necesitamos llamar al método *getConnection()* de la clase *DriverManager*. Tenemos estos tres métodos para realizar la conexión:

- *getConnection(String url):* utilizaremos este método cuando la URL contenga toda la información que necesitamos para conectarnos a la base de datos. Utilizaremos el *driver thin* para la conexión, con el nombre de usuario "ilerna", la contraseña "contrasena" y los datos de la base de datos.
  La clase *DriverManager* llamará al método *getConnection()* pasándole el *string* con la URL que hemos construido.

```
String url =
"jdbc:oracle:thin:ilerna/contrasena@localhost:1521:tema4";
Connection conn = DriverManager.getConnection(url);
if (conn != null) {
      System.out.println("Conexión realizada.");
}
```

- *getConnection(String url, Properties info):* a este método será necesario pasarle la URL con la información del *driver* de Oracle, el tipo de *driver* y el nombre de la base de datos. El resto de información se añadirá en una nueva instancia de la clase *Properties* que nos permitirá añadir los datos del usuario, la contraseña y además podremos definir con el *defaultRowPrefetch* o cuántas filas queremos que se devuelva cada vez del servidor. Este parámetro es opcional.

```
String url = "jdbc:oracle:thin:@tema4";
Properties properties = new Properties();
properties.put("user", "ilerna");
properties.put("password", "contrasena");
properties.put("defaultRowPrefetch", "20");

Connection conn = DriverManager.getConnection(url, properties);
```

- *getConnection(String url, String usuario, String contraseña):* en este método se pasan por parámetros los datos del usuario y la contraseña, además de la URL. La implementación sería algo más o menos así:

```
String url = "jdbc:oracle:thin:@tema4";
String usuario = "ilerna";
String pass = "contrasena";

Connection conn = DriverManager.getConnection(url, usuario, pass);
```

Para demostrar cómo se realizan todos los pasos, vamos a crear una clase auxiliar que se encargue de realizar el registro del *driver* y la conexión con la base de datos.

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

		Connection conn1 = null;
		Connection conn2 = null;
		Connection conn3 = null;

		try {
	// Se encarga de registrar el driver de oracle, es opcional
			Class.forName("oracle.jdbc.OracleDriver");
			// Metodo 1
			String url =
"jdbc:oracle:thin:ilerna/contrasena@localhost:1521:tema4";
			conn1 = DriverManager.getConnection(url);
			if (conn1 != null) {
				System.out.println("Conectado usando el metodo 1");
			}
			// Metodo 2
			String url2 = "jdbc:oracle:thin:@tema4";
			String usuario = "ilerna";
			String pass = "contrasena";
			conn2 = DriverManager.getConnection(url2, usuario, pass);
			if (conn2 != null) {
				System.out.println("Conectado usando el metodo 1");
			}

			// Metodo 3
			String url3 = "jdbc:oracle:thin:@tema4";
			Properties properties = new Properties();
			properties.put("user", "ilerna");
			properties.put("password", "contrasena");
			properties.put("defaultRowPrefetch", "20");

			conn3 = DriverManager.getConnection(url3, properties);

			if (conn3 != null) {
				System.out.println("Conectado usando el metodo 1");
			}
	} catch (ClassNotFoundException ex) {
			ex.printStackTrace();
	} catch (SQLException ex) {
			ex.printStackTrace();
	} finally {
			try {
				conn1.close();
				conn2.close();
				conn3.close();
			} catch (SQLException ex) {
				ex.printStackTrace();
			}
	}}

}
```

4.4. Características de las bases de datos orientadas a objetos

Otra base de datos importante es la base de datos orientada a objetos. Como hemos introducido anteriormente, una base de datos a objetos se define como:

La **base de datos orientada a objetos (BDOO)** es un tipo de base de datos mediante la cual representamos la información que contiene la base de datos mediante objetos.

Este tipo de base de datos nació para simplificar la programación orientada a objetos, y se combinó con las bases de datos tradicionales. El uso de una base de datos orientada a objetos simplifica la conceptualización porque, al usar objetos, permite representar de manera natural los datos que se quieren guardar.

Las **BDOO** implementan modelos conceptuales directamente y pueden representar complejidades que están más allá de las capacidades de los sistemas relacionales. Además, han adoptado muchos de los conceptos que se desarrollaron originalmente para el lenguaje de programación orientado a objetos.

Como características principales podemos destacar:

- **Soporta objetos complejos**: este tipo de base de datos se caracteriza por dar soporte a la construcción de objetos complejos aplicando constructores a objetos más básicos.
- **Encapsulamiento**: los programadores no tendrán acceso a la información, ya que este sistema gestor de BBDD orientado a objetos permite ocultar esos datos.
- **Identidad del objeto**: permite que todos los objetos tengan un identificador único que lo haga independiente de cualquier otro registro y le permita relacionarse con otros objetos.
- **Más rápidas**: son bases de datos mucho más rápidas porque no es necesario enlazar las tablas con *joins*.
- **Reduce el mantenimiento**: el principal objetivo de esta base de datos es mejorar el proceso de desarrollo y alargar la vida útil de la base de datos, reduciendo al mínimo su coste de mantenimiento. La gran mayoría de procesos están encapsulados y se pueden reutilizar e incorporarlos en nuevos procedimientos.
- **Modelo real de datos**: el sistema orientado a objetos tiende a transformar la información del mundo real de una manera más completa que los métodos tradicionales. Los objetos están organizados en clases de objetos, y están asociados con procesos. El modelo se basa en objetos más que en datos y procesamiento.

- **Fiabilidad y flexibilidad mejoradas**: el sistema orientado a objetos promete ser mucho más efectivo que los sistemas tradicionales, principalmente porque se pueden crear nuevos tipos a partir de objetos existentes. Debido a que los objetos se pueden llamar y acceder a ellos dinámicamente, se pueden crear nuevos objetos en cualquier momento. Los nuevos objetos pueden heredar atributos de datos de uno o de muchos otros objetos. Los comportamientos pueden heredarse de las superclases y pueden agregarse comportamientos nuevos sin afectar las funciones de los sistemas existentes.
- **Reutilización de código**: cuando se crea un nuevo objeto, heredará automáticamente los atributos de datos y las características de la clase a partir de la cual se generó. El nuevo objeto también heredará los datos y comportamientos de todas las superclases en las que participa.

Las bases de datos orientadas a objetos comparten muchos de los principios de la programación orientada a objetos, como son las siguientes características:

- **Encapsulación**: oculta información al resto de objetos para impedir conflictos o un acceso incorrecto.
- **Herencia**: jerarquía de clases a partir de las que los objetos heredan comportamientos.
- **Polimorfismo**: propiedad de una operación que permite aplicarse a objetos de distinta tipología.

4.5. Sistemas gestores de bases de datos orientadas a objeto (ODBMS, *object data base management system*).

Un **sistema gestor de bases de datos orientadas a objetos** es un sistema de gestión de bases de datos que soporta el modelo y la creación de información como objetos.

A diferencia de los sistemas de gestión de bases de datos relacionales, donde la información se guarda en tablas con filas y columnas, en las bases de datos orientadas a objetos se guarda como un objeto complejo y se establecen relaciones entre datos directamente, sin mapeos para relacionar las tablas y las columnas.

PL/SQL es otro tipo de lenguaje para bases de datos basado en objetos y funciones. Vamos a realizar una pequeña introducción para entender los conceptos básicos y los tipos de datos de este lenguaje.

El **lenguaje PL/SQL** (Procedural Language Extension of SQL) es un tipo de lenguaje enfocado a bases de datos basado en SQL y que combina este lenguaje junto con características de los lenguajes de programación.

Este lenguaje fue desarrollado por Oracle Corporation a principios de 1990 para mejorar las capacidades del lenguaje SQL. El código programado en este lenguaje se caracteriza por estar estructurado por bloques lógicos que pueden contener cualquier número de bloques anidados. PL/SQL está integrado en la base de datos Oracle desde su versión 7.

Las funcionalidades de PL/SQL generalmente se extienden después de cada lanzamiento de la base de datos Oracle. Aunque PL/SQL está estrechamente integrado con el lenguaje SQL, agrega algunas restricciones de programación que no están disponibles en SQL. Las funcionalidades más destacables de este lenguaje de programación son:

- Incluye condiciones y bucles.
- Declaración de constantes, variables, procedimientos, funciones y disparadores.
- Admite *arrays* y gestión de excepciones.
- Se puede usar SQL para manipular datos en Oracle.
- Se puede utilizar indistintamente mayúsculas o minúsculas para la creación de cualquier tipo de operaciones con este lenguaje, excepto para la gestión de cadenas de texto.

4.5.1. Gestores de bases de datos orientadas a objetos

En este apartado veremos en más profundidad el lenguaje PL/SQL. Este lenguaje está basado en SQL, por lo que es bastante fácil de entender y aprender.

En un programa PL/SQL, el código se escribe en bloques. El bloque PL/SQL crea los bloques lógicos estructurados de código que describen el proceso que ejecutar. Dicho bloque consta de sentencias SQL e instrucciones PL/SQL que luego se pasan al motor ORACLE para su ejecución. El bloque PL/SQL consta de las siguientes cuatro secciones:

- **DECLARE**: el código PL/SQL empieza con este bloque, donde se declararán las variables, objetos, triggers o cursores y, si es necesario, también pueden inicializarse. Una vez declarados o inicializados, podemos usarlos en

declaraciones SQL para la manipulación de datos. Esta sección es opcional, ya que nuestro código puede no necesitar declarar variables.

- **BEGIN**: en esta sección podremos encontrar las sentencias SQL y PL/SQL que deben ejecutarse, y contiene la lógica principal. Esta sección será la responsable de gestionar la consulta y manipulación de los datos. En esta parte se podrán introducir bucles, condicionales u otro tipo de operaciones típicas de la programación orientada a objetos. Esta sección es obligatoria.
- **EXCEPTION**: esta sección se encargará de gestionar y controlar los errores que se puedan producir entre los bloques BEGIN y EXCEPTION. Esta sección es opcional.
- **END**: esta sección es la que indica el final del bloque PL/SQL.

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(Hola! Es una prueba.');
END;
```

4.6. Tipos de datos: tipos básicos y tipos estructurados

En PL/SQL cada valor, ya sea de una constante, una variable o un parámetro, debe tener un tipo de dato que determinará el tipo de dato que guardar. Este lenguaje tiene dos tipos de datos básicos: *scalar* y *large object (LOB)*.

4.6.1. Tipos básicos

4.6.1.1. Scalar

Los tipos de datos *scalar* se subdividen en:

- *Number:* engloba todos los tipos de datos. Podemos encontrar subtipos de *number*. Los más importantes que podemos utilizar son:

Subtipo	Descripción
PLS_INTEGER	Valor numérico de tipo *integer* que puede estar en el rango -2.147.483.648 hasta 2.147.483.648 representado en 32 bits.
BINARY_INTEGER	*Integer* binario que puede estar en el rango -2.147.483.648 hasta 2.147.483.648 representado en 32 bits.
FLOAT	Tipo de dato *float* con un máximo de 126 dígitos binarios.
INT	*Integer*, no permite decimales con un máximo de 38 dígitos.
NUMBER	Número de punto fijo o de punto flotante con un valor absoluto en el rango 1E-130 a (pero sin incluir) 1.0E126. También puede representar el 0.
REAL	Soporta decimales *float* de un máximo de 63 bits.

Tabla 25. Tipos de datos de un objeto.

- *Boolean:* este tipo de dato soporta los valores *true*, *false* o *null*. Se utilizan normalmente en estructuras de control de flujo, como *if*, *case* o *loop*. Este tipo de datos no lo podemos encontrar en SQL, solo en tipos de lenguaje orientado a objetos.
- *Character:* engloba todos esos tipos de datos para la gestión de caracteres, como *char*, *varchar2*, *long*, *raw*, *long raw*, *rowid* y *unrowid*.

- *Datetime:* se utiliza para almacenar datos de tipo fecha. En la base de datos se guardan como un tipo numérico, por lo que da la posibilidad de trabajar con este tipo de datos con operaciones aritméticas.

4.6.1.2. *Large object* (LOB)

Son tipos de datos más complejos que ocupan mucho más tamaño que el resto de tipos de datos. Se consideran objetos grandes que se almacenan por separado de otros elementos de datos, como los caracteres, imágenes, vídeo y formas de onda de sonido. Este tipo de datos son como los ficheros binarios que tratábamos en el primer tema. De este tipo de datos podemos encontrar cuatro diferentes en PL/SQL:

- *BFILE:* se utiliza para guardar grandes ficheros binarios fuera de la base de datos.
- *BLOB:* se utiliza para guardar datos binarios de gran tamaño en la base de datos.
- *CLOB:* se utiliza para guardar cadenas muy largas de caracteres en la base de datos.
- *NCLOB:* se utiliza para almacenar grandes bloques de *NCHAR* en la base de datos. *NCHAR* es una cadena de caracteres de longitud fija de un máximo de 32.767 bytes.

Para declarar una variable con uno de estos tipos de datos, usaremos esta sintaxis:

```
nombre_variable [CONSTANT] tipoDato [NOT NULL] [:= | DEFAULT
valor_inicial]
```

Como podemos apreciar en el ejemplo, declarar una variable es bastante fácil. En primer lugar, debemos definir el nombre; si es una constante, deberemos añadir a continuación la palabra CONSTANT. Como es opcional, no es obligatorio ponerlo si no se trata de una constante. A continuación, debemos definir el tipo de dato y el tamaño. En el caso de la constante "pi" hemos inicializado la constante con un valor: para darle valor utilizamos dos puntos más el signo igual **(:=)**.

```
precio number(10, 2);
nombre varchar2(25);
dirección varchar2(233);
pi CONSTANT double precision := 3.1415;
```

4.6.2. Tipos estructurados: objetos en Oracle

En este apartado haremos una introducción al tipo de dato objeto.

Creación de objetos y métodos

Un objeto puede representar cualquier tipo de dato de la vida real. Por ejemplo, un dato real que representar podría ser un estudiante, un aula o un profesor. Para definir un objeto no podemos realizarlo dentro de un bloque PL/SQL, sino que tenemos que crearlo usando la sentencia SQL CREATE TYPE. Este tipo de dato encapsula una estructura de datos junto con funciones y procedimientos necesarios para manipular los datos definidos en él. Las variables que forman la estructura de datos se conocen como atributos. Las funciones y procedimientos que manipulan estos atributos se llaman métodos.

Una vez creado, se puede utilizar en un bloque PL/SQL para definir un tipo de una columna, un atributo, un elemento de una tabla o el resultado de una función dentro de un bloque. Vamos a ver un ejemplo:

```
CREATE TYPE nombre_objeto AS OBJECT (
   atributo         tipo_dato,
   atributo         tipo_dato,
   atributo         tipo_dato );
```

Para crear un objeto, debemos seguir esta sintaxis. Tal y como hemos mencionado, es necesario utilizar CREATE TYPE seguido del nombre que queramos dar al objeto y AS OBJECT para definir el tipo. A continuación, y entre paréntesis, irán los atributos de ese objeto y su tipo.

Dentro de un objeto podemos definir funciones y el valor que retornan, pero para poder utilizar esta función dentro de un objeto se tiene que crear, siguiendo esta estructura:

```
CREATE TYPE BODY nombre_objeto_donde_se_usara AS
MAP MEMBER FUNCTION nombre_funcion RETURN tipo_a_retornar IS
BEGIN
      RETURN nombre_atributo_objeto;
END;
```

La sintaxis que se utiliza para crear esta función es mucho más compleja y está envuelta en un bloque BEGUIN-END. Primero, con CREATE TYPE BODY definiremos el nombre del objeto que usará esta función, seguido de AS MAP MEMBER FUNCTION y el nombre de

la función junto con el tipo de dato que retornará. Dentro del bloque se definirá qué atributo de ese objeto retornará la función.

Vamos a realizar un pequeño ejemplo para entender el funcionamiento. En primer lugar, definiremos un objeto empleado:

```
CREATE TYPE empleado AS OBJECT (
   id_empleado NUMBER(6),
   nombre VARCHAR2(20),
   apellido VARCHAR2(20),
   MAP MEMBER FUNCTION get_id_empleado RETURN NUMBER);
```

Hemos definido una función dentro del objeto, la cual hará una llamada a una función que retorna un número. Vamos a realizar la creación de esta función:

Como vemos, es una simplificación de una función muy simple. Primero definiremos la función mediante CREATE TYPE BODY y añadiremos el objeto que usará esta función. AS MAP MEMBER FUNCTION nos permitirá añadir el nombre de la función que vamos a crear y definir el tipo de dato que va a retornar. Dentro del bloque BEGUIN, con el RETURN indicaremos qué atributo del objeto empleado vamos a retornar, en este caso, el id del empleado. Para terminar, tendremos que añadir la cláusula END.

```
CREATE TYPE BODY empleado AS
MAP MEMBER FUNCTION get_id_empleado RETURN NUMBER IS
BEGIN
      RETURN id_empleado;
END;
```

Declarar objetos

Podemos usar y declarar objetos dentro de los bloques de PL/SQL. EL procedimiento es bastante simple. Usaremos el bloque DECLARE para declarar el objeto añadiendo el nombre y el tipo de objeto, tal y como haríamos declarando un tipo simple.

```
DECLARE
  emp empleado; -- declaramos el objeto emp sin valor
BEGIN
-- call the constructor for employee_typ
  emp := empleado(315, 'Juan', 'Lopez');
  DBMS_OUTPUT.PUT_LINE(emp.nombre || ' ' || emp.apellido);
-- mostramos los detalles
  emp.get_id_empleado();  -- llamamos al método del objeto
END;
/
```

Como vemos en el ejemplo, se declarará el objeto añadiéndole el nombre y el tipo. Dentro del bloque BEGIN, inicializaremos el objeto haciendo una llamada al constructor y pasándole los valores de cada atributo por orden de definición.
En el ejemplo, imprimimos por pantalla el nombre del empleado y el apellido accediendo al *getter* de cada atributo: para ello, solo tenemos que poner el nombre del objeto, un punto y el nombre del atributo. Para llamar a un método se hará del mismo modo que hacemos en Java, con el nombre del método junto a paréntesis.

Manipulación de objetos

Si no inicializamos un objeto, el valor de este tipo de dato será *null*, es decir, el objeto en sí mismo, al declararlo, no tiene valor, es nulo, no solo sus atributos. Un objeto nulo nunca es igual a otro objeto. Siempre que comparemos un objeto nulo con otro objeto será nulo. Además, si asignamos un objeto nulo a otro objeto, se volverá automáticamente nulo.

Para acceder a un atributo de un objeto, accederemos a través del nombre y para acceder o cambiar el valor de un atributo, usaremos un punto. Si queremos acceder al objeto declarado dentro de un objeto, usaremos la concatenación de esos atributos para acceder al dato que nos interese. Por ejemplo:

```
CREATE TYPE jefe AS OBJECT (
   id_empleado NUMBER(6),
   nombre VARCHAR2(20),
   apellido VARCHAR2(20),
   emp empleado
);
```

Aquí, tenemos un nuevo objeto "jefe" que tiene como atributo un objeto "empleado". Para acceder y mostrar por consola los datos del empleado, accederemos a ellos mediante el nombre del primer objeto "jefe" seguido de un punto, el nombre del objeto al cual queremos acceder más el atributo: *jefe.emp.nombre*.

```
DECLARE
  jefe jefe; -- declaramos el objeto jefe sin valor
BEGIN
-- constructor de jefe
  jefe := jefe(315, 'Antonio', 'Lopez', empleado(2, 'Juan',
'Martinez'));
  DBMS_OUTPUT.PUT_LINE(jefe.emp.nombre || ' ' ||
jefe.emp.apellido);
-- mostramos los detalles
END;
/
```

En este ejemplo, vemos que tenemos que dar valor al objeto empleado declarado en "jefe", lo que se realiza llamando al constructor del objeto al que queremos dar valor.

Cuando pasamos parámetros a un constructor, la llamada asigna inicialmente los valores a los atributos del objeto inicializado. Cuando usamos el constructor predeterminado, tenemos que añadir valor a cada uno de los atributos del objeto siguiendo el estricto orden de creación y el tipo de dato que necesiten. Es decir, si tenemos que dar valor a un *number*, tenemos que pasar un *number* y no un *varchar2*.

A diferencia de las constantes y variables, los atributos no pueden tener valores predeterminados. Puede llamar a un constructor usando notación con nombre en lugar de notación posicional.

En las declaraciones SQL, las llamadas a un método sin parámetros requieren una lista de parámetros vacía. En las declaraciones de procedimiento, una lista de parámetros vacía es opcional a menos que encadene llamadas, en cuyo caso se requiere para todas menos para la última llamada. No puede encadenar llamadas de método adicionales a la derecha de una llamada a procedimiento porque un procedimiento se llama como una declaración, no como parte de una expresión. Además, si encadena dos llamadas de función, la primera función debe devolver un objeto que se pueda pasar a la segunda función.

4.7. Definición y modificaciones de objetos. Consultas y gestión de transacciones

En este apartado nos centraremos en la gestión de los objetos y cómo realizar modificaciones, eliminación y consultas de objetos en el lenguaje PL/SQL.

4.7.1. Definición y modificación de objetos

Tanto en PL/SQL como en SQL las sentencias que modifican los datos de una base de datos son INSERT, UPDATE o DELETE. Tendremos que realizar cualquiera de estas tres operaciones dentro del bloque BEGIN, y podremos realizar tantas operaciones como deseemos.

Estas sentencias son las que utilizaremos para realizar modificaciones a los objetos que hayamos creado.

Para poder entender las diferentes acciones que os vamos a explicar, primero tenemos que tener creados objetos para poder explicar el proceso. Crearemos un objeto producto con la sintaxis CREATE OR REPLACE TYPE. Lo que hará es crear el objeto o sustituirlo si ya existe.

```
CREATE Or Replace TYPE ProductoType AS OBJECT (
  id          NUMBER,
  nombre        VARCHAR2(15),
  descripcion VARCHAR2(22),
  precio       NUMBER(5, 2),
  dias_validez  NUMBER
);
```

A continuación, crearemos una tabla a la que llamaremos "productos" que almacenará un contador y un objeto de tipo *ProductoType*.

```
CREATE TABLE productos (
    producto           ProductoType,
    contador             NUMBER
);
```

Ahora que tenemos la base de datos preparada, para realizar un INSERT en una tabla donde una de las columnas es un objeto, utilizaremos esta sintaxis:

```
INSERT INTO productos (producto,contador) VALUES (ProductoType(1,
'AA', 'BBB', 3.95, 10),50);
```

La estructura del INSERT es la misma que usamos en SQL, tenemos que usar INSERT INTO más el nombre de la tabla y entre paréntesis los atributos de esa tabla. En el caso de los objetos, esto no cambia, se añade el nombre del objeto tal y como está definido en la base de datos. Seguidamente, se añadirá la cláusula VALUES y entre paréntesis se empezarán a añadir los valores de cada columna. Para los objetos tendremos que crear uno nuevo. Para ello, se creará usando el constructor de *ProductoType* y pasándole cada atributo por parámetro. Al terminar, se añadirá una coma para separar una columna de otra, como en SQL.

Para realizar una modificación de un registro, usaremos la cláusula UPDATE, tal y como realizamos en SQL. Para actualizar un objeto, usaremos esta sintaxis:

```
UPDATE productos p set p.producto.descripcion='Nueva descripcion'
WHERE p.producto.id = 1;
```

Para modificar el registro de la tabla "productos", primero añadiremos un alias a esa tabla, lo que nos servirá para acceder a cada columna usando este alias. Es especialmente útil cuando hay relaciones con más de una tabla. Como vemos, para acceder al objeto usamos *p.producto*. Se utiliza el punto para anidar la información a la que queremos acceder. En este caso, queremos actualizar la descripción del objeto "producto", para indicar qué columna cambiar escribiremos *p.producto.descripcion* y asignaremos el nuevo valor. Si en lugar de querer modificar la descripción quisiéramos cambiar el precio, lo indicaríamos de este modo: *p.producto.precio*. El valor que asignemos a la columna debe corresponder con su tipo, no podemos asignar un *number* a una columna que tenga un *varchar2*, ya que los tipos no coinciden y se produciría un error.

Para eliminar un registro, también se utiliza la misma sintaxis que SQL, de este modo:

```
DELETE FROM productos p WHERE p.producto.id = 1;
```

Como podemos apreciar en el ejemplo, para realizar la eliminación tendremos que borrar el registro de la tabla "productos" y eliminará el registro.

4.7.2. Consultas

Seleccionar registros de una tabla que contenga una columna de tipo objeto es bastante sencillo, ya que se siguen utilizando las sentencias que ya conocemos de SQL.

```
select producto FROM productos;
```

La sintaxis que tenemos que utilizar es bastante parecida a SQL. Para seleccionar todas las columnas de tipo objeto de la tabla "productos", realizaremos un SELECT sobre el nombre de esa columna. La diferencia respecto a una selección normal es que por consola podremos ver cada registro de tipo objeto y todos sus contenidos. Por consola tendremos que ver algo parecido a esto:

```
  PRODUCT(ID, NOMBRE, DESCRIPCION, PRECIO, DIAS_VALIDEZ)
-------------------------------------------------------
ProductoType(1, 'AA', 'new D', 3.95, 10)
ProductoType(2, 'CC', 'DDDD', 2.99, 5)
```

El resultado es una lista de registros de tipo objeto que entre paréntesis muestra los atributos de cada objeto.

4.7.3. Gestión de transacciones

Como hemos visto en apartados anteriores, una transacción es una secuencia de operaciones realizadas mediante la ejecución de las declaraciones SQL dentro de un bloque de código PL/SQL, en este caso.

Las transacciones tienen que seguir estas reglas:

- Si tenemos un bloque de sentencias en una transacción y se ejecutan en bloque pero una de ellas falla, tendrá efecto sobre todas las sentencias que se han ejecutado previamente.
- En una transacción, todo el bloque de sentencia se tiene que ejecutar sin fallos para que se pueda ejecutar el *commit;* si no es así, se ejecuta el *rollback*.

- El alcance de una transacción se define mediante los comandos *commit* o *rollback*.
- Una transacción finaliza cuando se produce uno de los siguientes eventos: un *commit*, cuando se han terminado todas las acciones sin error, o un *rollback*, que se ejecuta cuando se ha producido un error y se necesita tirar atrás todas las acciones realizadas.

Una transacción empezara en el momento que se encuentre la primera sentencia SQL y termina cuando se ha ejecutado el *commit* o el *rollback*.

El *commit* será la última sentencia de una SQL antes de la cláusula END. Se ejecutará después de cualquier comando DML. La sintaxis será así:

```
Commit;
```

Por otro lado, el *rollback* se ejecutará cuando alguna de las sentencias del bloque PL/SQL falle, ya sea por mala estructura o por algún otro tipo de fallo.

El comando *rollback* no tendrá ningún efecto si se ejecuta después del comando *commit* porque en ese caso el comando *commit* hará que los cambios realizados en la transacción sean permanentes.

Se utilizará la siguiente sintaxis:

```
Rollback [to savepoint <nombre_savepoint >];
```

Savepoint es un parámetro opcional que se puede añadir y se utiliza para hacer *rollback* hasta un punto especificado. El *nombre_savepoint* hará referencia al nombre dado al *savepoint* creado durante la transacción, y está definido por el usuario.

Los *savepoints* son especialmente útiles cuando tenemos una transacción larga, ya que se encarga de dividir esta transacción en transacciones más pequeñas y marca ciertos puntos con el *savepoint* como puntos de control. Nos será útil cuando queramos revertir una parte particular de la transacción en lugar de hacer *rollback* de todas las partes.

Por ejemplo, si una transacción completa tiene ocho sentencias y creamos un *savepoint* después de cuatro sentencias, si por alguna razón después de la ejecución de la sexta sentencia queremos revertir la cuarta sentencia, entonces podemos hacer eso fácilmente y la transacción puede ejecutarse nuevamente a partir de la cuarta instrucción.

Para la creación de un *savepoint*, debemos usar esta sintaxis:

```
Savepoint <nombre_savepoint >;
```

Vamos a ver un ejemplo de cómo implementar un *commit*, un *savepoint* y un *rollback:*

```
DECLARE
      snm estudiante.nombre%type;
      s_edad estudiante.edad%type;
      s_apellido estudiante.apellido%type;
BEGIN
      snm := '&nombre';
      s_edad := &edad;
      s_apellido := '&apellido';
      INSERT into estudiante values(snm,edad,apellido);
      dbms_output.put_line('Se ha añadido un registro');
      COMMIT;
      -- añadimos savepoint
      SAVEPOINT ejemplo_savepoint;
      -- Se piden datos por segunda vez
      rollno := &sno;
      snm := '&nombre';
      s_edad := &edad;
      s_apellido := '&apellido';
      INSERT into estudiante values(snm,edad,apellido);
      dbms_output.put_line('Otro valor que se añade');
      ROLLBACK [TO SAVEPOINT ejemplo_savepoint];
END;
```

4.8. La interfaz de programación de aplicaciones de la base de datos

En este apartado aprenderemos a ejecutar bloques de código PL/SQL en Java. Principalmente, nos centraremos en sentencias orientadas a objetos. Como hemos visto en el apartado 4.3, para desarrollar nuestra aplicación, añadiremos la librería OJDBC para interactuar con la base de datos. Esta librería añade la posibilidad de trabajar con una base de datos de Oracle, pero tiene todas las funcionalidades JDBC que hemos visto en temas anteriores.

Sentencias para creación, modificación, eliminación e inserción

Para la **creación, modificación, eliminación e inserción** de objetos PL/SQL podemos usar la interfaz *Statement*, tal y como introdujimos en el tema 2. Este tipo de objeto se utiliza para conexiones de carácter general y no acepta parámetros. Es bastante útil cuando queremos usar consultas estáticas SQL. Las sentencias más utilizadas son todas aquellas que no necesitan datos de los objetos Java. En nuestro caso, solo queremos ejecutar una sentencia y, como nuestra base de datos es apta para PL/SQL, con la interfaz *Statement* podemos realizar esta tarea. Vamos a mostrar un ejemplo de implementación:

```
Statement stmt= null;
ConexionOracle aux = new ConexionOracle ();
Connection conn = aux.realizarConexion();
try {
	stmt = conn.createStatement();

	String sql;
	sql = "CREATE Or Replace TYPE ProductoType AS OBJECT"+
		" (id NUMBER,"+
		" nombre        VARCHAR2(15),"+
		"  descripcion VARCHAR2(22),"+
		" precio       NUMBER(5, 2),"+
		" dias_validez  NUMBER"+
		");";
	stmt.execute(sql);

	sql = "CREATE TABLE productos (" +
		"producto           ProductoType," +
		" contador             NUMBER    );";
	stmt.execute(sql);
} catch (SQLException e) {
	e.printStackTrace();
} try {
	stmt.close();
	conn.close();
} catch (SQLException e) {
	e.printStackTrace();
}
```

En primer lugar, tenemos que declarar todos los objetos que tenemos que utilizar: primero, el *Statement*, que servirá para ejecutar las sentencias; en segundo lugar, *ConexionOracle*, que es la clase auxiliar que establecerá la conexión con la base de datos; y por último *Connection*, que recibirá los datos de la conexión.

Después de declarar los objetos que utilizaremos, vamos a inicializar la instancia de la interfaz *Statement* con ayuda de la instancia *Connection* y la llamada al método *createStatement()*. Una vez realizados todos estos pasos, ya podemos crear en un *string* nuestras sentencias PL/SQL. Podremos realizar todo tipo de acciones con esta estructura, desde crear tablas y objetos hasta eliminar, actualizar o introducir datos. El proceso para estas acciones será siempre con esta estructura. Para ejecutar la

operación, solo tenemos que utilizar *stmt.execute()* y le pasaremos la consulta por parámetro. Si no se produce ningún error, la operación habrá terminado y solo nos quedará cerrar los objetos abiertos.

Debemos tener en cuenta que, cada vez que creemos este tipo de objetos, tendremos que cerrarlos. Por eso, es necesario utilizar el método *close()* para cerrar la conexión.

Selección de objetos

Para la selección de objetos de la base de datos Oracle, en esta ocasión también usaremos la interfaz *Statement*, junto con *ResultSet* y *Struct*.

En JDBC existe una interfaz para realizar operaciones con PL/SQL. Se trata de la interfaz *Struct* que podemos encontrar en el paquete java.sql.*.

Antes de empezar con la implementación, debemos estudiar los métodos que podremos usar y que nos serán de utilidad a la hora de implementar el código:

Método	Descripción
getSQLTypeName()	Este método se encarga de devolver el tipo del objeto devuelto.
getAttributes()	Este método devuelve los atributos del objeto de la base de datos, en el orden en que se han creado.

Tabla 26. Métodos más importantes de la interfaz de java.sql.Struct.

En el ejemplo siguiente, podremos ver qué estructura debemos seguir si queremos realizar una selección de objetos en la base de datos:

```
Statement stmt = null;
aux = new ConexionOracle();
conn = aux.realizarConexion();
try {
      stmt = conn.createStatement();
      String sql = "select producto FROM productos; ";
      ResultSet rs = stmt.executeQuery(sql);
      while (rs.next()) {
            Struct productos = (Struct) rs.getObject(1);
            Object[] elements = productos.getAttributes();
            int id = (int) elements[0];
            String nombre = (String) elements[1];
            String descripcion = (String) elements[2];
            int precio = (int) elements[3];
            int validez = (int) elements[4];

            System.out.println("Id Producto: " + id);
            System.out.println("Nombre: " + nombre);
            System.out.println("Descripcion: " + descripcion);
            System.out.println("Precio: " + precio);
            System.out.println("Validez: " + validez);
      }
      rs.close();
} catch (SQLException e) {
      e.printStackTrace();
}
try {
      stmt.close();
      conn.close();
} catch (SQLException e) {
      e.printStackTrace();
}
```

En primer lugar, tenemos que declarar todos los objetos que tenemos que utilizar: primero, el *Statement*, que servirá para ejecutar las sentencias; en segundo lugar, *ConexionOracle*, que es la clase auxiliar que establecerá la conexión con la base de datos; y por último, *Connection*, que recibirá los datos de la conexión.

A continuación, tendremos que definir en un *string* la consulta que queramos realizar. En esta ocasión, haremos una selección de productos que hemos creado en el ejemplo anterior. Para ejecutar la consulta, crearemos una nueva instancia de la interfaz *ResultSet*. Esta interfaz también la hemos explicado con detalle en el tema 2. Mediante la instancia de *Statement* y la llamada a *executeQuery()* llenaremos con los resultados el objeto *ResultSet*. El objeto *ResultSet* tiene un método que nos permitirá recorrer los resultados uno a uno y tratar los datos obtenidos, el método *.next()*.

Como vemos en el ejemplo, podremos acceder a los datos y tratarlos de la manera que deseemos. Para transformar el primer resultado que encuentre el bucle, tenemos que

transformarlo a una nueva instancia de *Struct* haciendo un *cast* de *rs.getObject(1)*. A continuación, tendremos un *array* con todos los elementos de cada fila, y los añadiremos a un *object[]* con el uso del método *getAttributes()*. Para recorrer y obtener cada columna, accederemos mediante *elements[]* y empezaremos desde el 0 hasta 4, ya que tenemos 5 columnas, y asignaremos a cada tipo de dato que corresponda todos los elementos de la fila e iremos accediendo a cada elemento según el orden en que se encuentren en la base de datos. Por consola iremos mostrando la información obtenida en la búsqueda.

4.9. Prueba y documentación de aplicaciones desarrolladas

Una de las partes más importantes que tener en cuenta a la hora de programar son las pruebas y la documentación. En este apartado vamos a poner en práctica unos cuantos ejercicios para aprender a usar PL/SQL, como dejar bien comentado el código en este lenguaje y las buenas prácticas que seguir.

En primer lugar, nos centraremos en cómo realizar la documentación de la clase DAO que hemos creado en el apartado anterior.

```
/**
 * Clase que se encargara de realizar todas las operaciones con la base de datos.
 * @author Laura Gatius
 *
 */
public class EjercicicosDaoImpl {

    ConexionOracle aux = null;
    Connection conn = null;

    public void creacionTablasObjetos() {
        Statement stmt = null;
        aux = new ConexionOracle();
        conn = aux.realizarConexion();
        try {
            stmt = conn.createStatement();

            String sql;
```

Ilustración 39. Ejemplo de cómo documentar la clase.

Tenemos la clase "EjerciciosDaoImpl", esta clase la hemos creado para realizar las operaciones con la base de datos. Esta información, que es un resumen de la función de la clase, es lo que tendremos que añadir como JavaDoc. Si revisamos el tema 1, tenemos una breve explicación de cómo realizar la documentación y de las etiquetas y puntos más importantes. Siempre que se desee, se tendrá que añadir la anotación *author*, que indicará quién ha realizado esa clase.

A continuación, tenemos que añadir a la documentación todos los métodos que existan en esa clase. El procedimiento es el mismo, y se tiene que añadir un breve resumen de la función principal de ese método. Tendría que quedar algo parecido a esto:

```
/**
* Método que realiza la creación de un objeto y de una tabla en
la base de datos Oracle. Retorna true si ha ido bien.
 * @return result.
*/
public boolean creacionTablasObjetos() {
	Statement stmt = null;
	aux = new ConexionOracle();
	conn = aux.realizarConexion();
	boolean result = true;
	try {
		stmt = conn.createStatement();

		String sql;
		sql = "CREATE Or Replace TYPE ProductoType AS OBJECT
(id NUMBER," + " nombre        VARCHAR2(15),"
		+ "  descripcion VARCHAR2(22)," + " precio
NUMBER(5, 2)," + " dias_validez  NUMBER" + ");";

		stmt.execute(sql);
		sql = "CREATE TABLE productos (" +
		"    producto           ProductoType,"
		+ "    contador             NUMBER);";
		//Creción de la sentencia
		stmt.execute(sql);

	} catch (SQLException e) {
		result = false;
		e.printStackTrace();
	}
	try {
		stmt.close();
		conn.close();
	} catch (SQLException e) {
		result = false;
		e.printStackTrace();
	}
	return result;
}
```

Después de comentar el método, tenemos que añadir algún comentario dentro del método comentando qué hacen las partes más importantes del código para que quede claro para qué sirve si otro programador revisa el código.

```
/**
* Método que realiza la creación de un objeto y de una tabla en
la base de datos Oracle. Retorna true si ha ido bien.
 * @return result.
*/
public boolean creacionTablasObjetos() {
	//Creación de las instancias Statement y de la conexión
	Statement stmt = null;
	aux = new ConexionOracle();
	conn = aux.realizarConexion();
	boolean result = true;
	try {
		//Damos valor a la interfaz Statement
		stmt = conn.createStatement();

		String sql;
		sql = "CREATE Or Replace TYPE ProductoType AS OBJECT
(id NUMBER," + " nombre        VARCHAR2(15),"
		+ "  descripcion VARCHAR2(22)," + " precio
NUMBER(5, 2)," + " dias_validez  NUMBER" + ");";
		//Creción de la sentencia
		stmt.execute(sql);
		sql = "CREATE TABLE productos (" +
		"    producto           ProductoType,"
		+ "    contador             NUMBER);";
		//Creción de la sentencia
		stmt.execute(sql);

	} catch (SQLException e) {
		result = false;
		e.printStackTrace();
	}
	try {
		//Cerramos conexión y statement
		stmt.close();
		conn.close();
	} catch (SQLException e) {
		result = false;
		e.printStackTrace();
	}
	return result;
}
```

En segundo lugar, nos centraremos en realizar los test de la clase DAO que hemos creado en el apartado anterior. Crearemos una nueva clase *test* que probará las operaciones que hemos comentado en el apartado anterior. Para ello, crearemos una clase a la que llamaremos "ProductosTest" en nuestro proyecto. En ella testearemos todos los métodos de la clase "EjerciciosDaoImpl". Esto nos servirá para comprobar que realmente funciona y evitar posibles errores. Es especialmente útil para no subir a producción un código erróneo y comprobará la calidad del código.

Primero realizaremos el test del método *creacionTablasObjetos():*

```
/**
* test que comprueba si el método creacionTablasObjetos() de
* EjerciciosDaoImpl
* funciona correctamente
*/
@Test
public void testCreacion() {
	boolean b = true;
	EjerciciosDaoImpl dao = new EjerciciosDaoImpl();
	//Si funciona bien, dará true.
	assertTrue(dao.creacionTablasObjetos());
}
```

Tal y como vemos en este ejemplo, el test es bastante sencillo. Haremos una nueva instancia de la clase "EjerciciosDaoImpl" que nos servirá para que la conexión se realice y para poder usar los métodos que hemos creado en esa clase. De este modo, podremos probar si funcionan o no. Utilizaremos *assertTrue* para comprobar que el método *creacionTablasObjetos()* se ejecuta correctamente. Si todo va bien, devolverá *true*; si no, retornará *false*. Si nos da *false*, tendremos que revisar el funcionamiento de ese método y corregir los errores que puedan surgir hasta que el test dé *true*.

Este procedimiento tendremos que realizarlo para todos los métodos. Recordad que no es necesario utilizar siempre *assertTrue*, sino que hay más posibilidades que ya explicamos durante el tema 1.

4.10. Lenguaje de consultas para objetos (OQL, *object query language*)

El **lenguaje de consultas para objetos (OQL)** es un tipo de lenguaje inspirado en SQL y que, en lugar de usar el nombre de las tablas, usa el nombre de los objetos del lenguaje orientado a objetos que se use.

El lenguaje de consulta más utilizado por la mayoría de herramientas es el lenguaje llamado OQL, un lenguaje especificado por el ODMG (*object data management group*).

Presenta cierta similitud con SQL, dado que ambos son lenguajes de consulta no procedimental, pero el OQL está totalmente orientado a objetos, es decir, los componentes de la consulta se expresan utilizando la sintaxis propia de los objetos y los resultados obtenidos devuelven objetos o colecciones de objetos.

Se trata del lenguaje de consulta que también usan muchas bases de datos orientadas a objetos, lo que lo convierte en uno de los estándares más populares y conocidos.

Las características principales de este tipo de lenguaje son:

- Se usan los mismos operadores:

Operador	Descripción
=	Para comparar valores iguales.
<>	Para indicar que un valor no es igual a.
<	Menor que.
>	Mayor que.
<=	Menor o igual que.
>=	Mayor o igual que.
like	Compara la similitud utilizando expresiones regulares de estilo UNIX. El operador *like* es más poderoso que las implementaciones SQL tradicionales que utilizan la coincidencia de testigo más limitada para la comparación.
in	Busca la presencia de un registro dentro de una tabla especificada (utilizando una subconsulta). El operador *in* también se puede utilizar para determinar si hay un valor especificado en un campo de la lista.
es nulo	Comprueba si el valor de la columna es nulo.
no es nulo	Comprueba si el valor de la columna no es nulo.

Tabla 27. Operadores más usados.

- Las sentencias, tal y como ocurre en SQL, se pueden escribir tanto en mayúsculas como en minúsculas.
- Reglas generales de SQL: finalizar query con punto y coma **(;)**, cadenas de texto entre comillas dobles (siempre) y la lista de entradas separadas por una coma.
- Puntuación en sentencia: OQL permite símbolos de puntuación en las sentencias. Los principales son estos:

Símbolo	Descripción
-	Para representar símbolos de resta o negativos.
+	Para representar símbolos de suma o positivo.
*	Para indicar la selección de todos los registros, o como símbolo de multiplicación.
(	Abrir paréntesis para las sentencias UPDATE o INSERT o encerrar elementos de una lista.
)	Cerrar paréntesis para las sentencias UPDATE o INSERT o encerrar elementos de una lista.
,	Separador de listas o condiciones.
[ ]	Para agrupar elementos de una lista.
{ }	Para agrupar elementos tipo de datos del objeto.
.	Para separar el nombre de la base de datos del nombre de la tabla. También en columnas.
;	Para finalizar la sentencia OQL.
~	Para expresiones regulares, significa que los valores coinciden.
->	Se utiliza en los objetos para recuperar un subvalor.

Tabla 28. Símbolos más utilizados y aptos para las sentencias OQL.

- Facilidad de aprendizaje: es muy similar a SQL, las diferencias son mínimas.

4.10.1. Crear bases de datos o tablas

EL lenguaje OQL permite la creación de bases de datos y tablas, tal y como hemos visto en SQL. La estructura para crear una base de datos seria:

```
create database nombre_base_datos;
```

La sentencia es prácticamente igual que SQL, es necesario definir *create* para definir que queremos realizar la operación de crear seguido de *datatable*, que es la palabra definida para crear una base de datos nueva. Seguidamente, añadiremos en nombre de la base de datos y finalizaremos con punto y coma.

Para la creación de tablas, se sigue una estructura similar a SQL. Se define como *create table* seguido del nombre de la base de datos junto a un punto y el nombre de la tabla que queremos crear. Entre paréntesis definiremos las diferentes columnas seguido del tipo de datos y si es un valor obligatorio o no. Este dato se añade añadiendo NULL o NOT NULL después del tipo de datos. A diferencia de SQL, no se añade el tamaño del tipo de datos, solo el tipo.

```
create table nombre_BBDD.nombre_tabla (
   nombre_columna [ constraints ] [ default default ] ,
   [ nombre_columna [ constraints ] [ default default]
);
```

Tipos de datos

Los datos que se utilizan para la creación de tablas son algo distintos de los que se utilizan en SQL. Aquí os mostramos los más importantes y para qué se utilizan:

Tipo de datos	Descripción
TEXT	Contiene texto sin formato.
INT	Para definir números sin decimales.
UINT	Contiene un valor entero sin signo de 32 bits.
FLOAT	Para definir números con decimales.
LONG64	Contiene un valor numérico de 64 bits.
ULONG64	Contiene un valor numérico sin signo de 64 bits.
DATA	Para definir datos de tipo binario.
LIST TYPE *datatype*	Representa una lista de datos concretos y se engloba entre corchetes [].
OBJECT TYPE *datatype*	Este será el tipo de dato objeto y se encerrará entre llaves {}. Dentro del objeto tendremos parejas de nombre valor, en una lista.
TIME	Se utiliza para definir datos referentes a fechas o tiempo.

Tabla 29. Tipo de datos soportados por OQL.

4.10.2. Inserción

A continuación, vamos a detallar cómo se realiza un INSERT en el lenguaje OQL. El proceso es sencillo y similar a SQL, la única diferencia destacable es que para definir en qué tabla se realiza la inserción se tiene que indicar el nombre de la base de datos junto con un punto y la tabla. La sentencia sigue siendo INSERT INTO. Se definirán las columnas de esa tabla y se añadirán todos los valores con el orden de las tablas para que se inserten en el orden deseado. En OQL, es aceptado omitir el nombre de la columna, aunque se recomienda añadirlo porque no poner las columnas se considera mala práctica. Si se decide no añadir el nombre de las columnas, el usuario también tendrá que seguir el orden de las columnas de la base de datos. A las columnas que no se desee añadir un valor se les puede añadir un NULL. La estructura será la siguiente:

```
insert into nombre_BBDD.nombre_tabla ( columna, columna,
columna,…)
values (datos, datos, datos datos);
```

4.10.3. Selección

Para la selección tenemos opciones diferentes en OQL: SELECT y SELECT INTO. Para empezar, la más básica es la operación de selección. Es la más utilizada y se asemeja a SQL. La estructura que sigue es esta:

```
select lista_valores o * (para todos los valores)

from nombre_BBDD.nombre_tabla where condicion order by
nombre_columna [asc|desc] (opcional);
```

Tal como ocurre en SQL, se puede escoger una selección de columnas separadas por coma, o usar el asterisco para devolver todas las columnas de la tabla. El *order by* también se puede usar en OQL y ordenará la columna que indiquemos en el orden ascendente o descendente, según queramos.

Como novedad, OQL introduce un tipo de selector especial, el SELECT INTO. Se encargará de seleccionar datos e introducirlos en otra tabla. Realiza dos acciones en una sola ejecución. Los datos que se seleccionan no desaparecen de la tabla de origen, solo se añaden a una tabla destino.

```
select lista_valores o * (para todos los valores)
into nombre_BBDD_destino.nombre_tabla_destino
from nombre_BBDD.nombre_tabla
where condicion;
```

La selección de esta sentencia se introducirá a la tabla destino en el orden seleccionado, independientemente de la estructura y las columnas de la tabla destino. El WHERE en esta ocasión también es una cláusula opcional. Esta sentencia, aparte de lo mencionado, es bastante más completa que una SQL normal porque es capaz de detectar si uno de los valores seleccionados es nulo y saltarse ese registro. De este modo, evitamos registros nulos en la base de datos durante esta sentencia.

4.10.4. Modificar registros

Para modificar registros, en OQL tenemos la sentencia UPDATE, tal y como conocemos mediante SQL. Esta sentencia se usará para actualizar datos de la base de datos según unos criterios que añadiremos mediante el WHERE.

```
update nombre_BBDD_destino.nombre_tabla_destino
set columna = valor, columna=valor, …
where condicion;
```

Como podemos apreciar, la sentencia es prácticamente igual y se utiliza del mismo modo.

4.10.5. Eliminar registros

Para eliminar registros usaremos esta sintaxis:

```
delete
from nombre_BBDD.nombre_tabla
where condicion;
```

El procedimiento es el mismo que en SQL, no se pueden apreciar cambios de OQL a SQL. El WHERE es opcional, pero es recomendable añadirlo para evitar que se borren todos los datos de la tabla.

OQL, además, tiene el mandato DROP, que a diferencia de SQL puede borrar tanto una tabla como una base de datos. La sintaxis que sigue es esta:

```
drop table nombre_BBDD.nombre_tabla ;
drop datatable nombre_BBDD ;
```

4.10.6. Subconsultas

En lenguaje OQL están permitidas las subconsultas. Tal y como el nombre indica, son consultas dentro de consultas y están anidadas bajo corchetes ((). Dentro de los corchetes se pueden incluir todas las sentencias, no existe restricción. Vamos a ver la sintaxis de una subconsulta:

```
select * from nombre_BBDD.nombre_tabla where condicion in (
(select * from nombre_BBDD.nombre_tabla where condicion));
```

Así sería una consulta dentro de una consulta. Su funcionamiento es igual que en SQL, la única diferencia es que, en lugar de englobarse en un paréntesis de apertura y otro para cerrar, en OQL se usa doble paréntesis.

En definitiva, a grandes rasgos, las principales diferencias que podemos observar ente SQL y OQL es que:

1. OQL soporta tipo objeto en las sentencias.
2. No todas las palabras clave de SQL se pueden utilizar en OQL.
3. OQL puede realizar cálculos matemáticos en las sentencias, en cambio, SQL no lo contempla.
4. SQL no está orientado a objetos como OQL, es más de tipo relacional.

5. Bases de datos XML

En este tema, haremos una introducción a las bases de datos XML.

Una base de datos XML es un sistema de persistencia que guarda la información en formato XML.

Este tipo de base de datos utiliza un lenguaje de consulta propio llamado XQuery.

El lenguaje XQuery es un tipo de lenguaje de consulta establecido para bases de datos XML. Está diseñado para realizar consultas sobre colecciones de datos.

Existen dos tipos de bases de datos XML:

- Base de datos activada.
- Base de datos nativa.

5.1. Bases de datos nativas XML. Comparativa con base de datos relacional. Ventajas e inconvenientes.

Una **base de datos nativa** (NXD) es aquel sistema de base de datos que se caracteriza por guardar los datos en documentos XML.

Este tipo de base de datos se caracteriza por ser un modelo lógico para ficheros XML, y se centra en recuperar y almacenar los documentos siguiendo este modelo. En este tipo de base de datos se utiliza el XPath para realizar las consultas.

El XPath es un tipo de lenguaje utilizado en XML para el acceso a la información del documento, recorriendo los elementos del fichero XML.

El lenguaje XPath se utiliza para consultar los datos de un XML, se encarga de establecer una expresión y verificar si hay coincidencias en el documento que cumplan los

requisitos. Este tipo de lenguaje no se utiliza solo en lectura de XML, sino también en consultas XQuery o en transformaciones XSLT.

Este tipo de base de datos nos ofrece un tipo de características que no podemos encontrar en otro tipo de bases de datos. Sus características principales son:

- **Modelo lógico**: siguen un modelo lógico para almacenar el documento XML.
- **Validación**: una base de datos XML permite la validación de los documentos a partir de la introducción de documentos XSD o DTD. Estos documentos definen la estructura que debe tener un XML para ser válido.
- Recupera y almacena la información siguiendo este modelo.
- El modelo incluye elementos y atributos y debe permitir la gestión de PCDATA *(parsed character data)*. Estas siglas significan que un documento contendrá texto.
- Conservación de estructura del XML: este tipo de base de datos conserva la estructura del documento.
- Permite lenguajes de consulta XML.
- **Consulta de datos**: soporta más de un lenguaje de consulta. Uno de los más populares es XPath o XQuery.
- **Estructura con nodos**: se representan como nodos los elementos, atributos, instrucciones de procesamiento, comentarios y cualquier otro elemento constituyente de un documento XML.
- **Orden**: preserva el orden del documento, las instrucciones de procesamiento, los comentarios, las secciones PCDATA y las entidades.
- **Flexibilidad**: permite almacenar cualquier tipo de documento XML que esté validado, sin la necesidad de tener un modelo previo ligado a cada tipo de documento XML que se quiera almacenar.
- No tiene ningún modelo de almacenamiento físico concreto. Este tipo de base de datos se puede construir sobre una base de datos de tipo relacional, jerárquica, orientada a objetos o de tipo propietario. Recordemos que una base de datos de tipo propietario es aquella cuyo código está cerrado y su uso y distribución está también limitado, es decir, el propietario de esa base de datos es una empresa que se encarga de distribuir el producto ya sea con o sin coste, por ejemplo, MySql o Firebird.
- Almacenamiento de documentos en colecciones:
    - Las colecciones juegan en las bases de datos nativas el papel de las tablas en las bases de datos relacionales.
    - Los documentos se suelen agrupar, en función de la información que contienen, en colecciones que a su vez pueden contener otras colecciones.
- Indexación: este tipo de base de datos permite la creación de índices que aceleran las consultas que se realizan de forma más frecuente.

- Identificadores únicos: cada elemento XML tiene asociado un identificador único por el que será reconocido dentro del repositorio.
- Actualizaciones y borrados: estas bases de datos tienen gran variedad de estrategias para actualizar y borrar documentos. Una de estas estrategias es XUpdate, es un tipo de lenguaje especializado en la realización de modificaciones en la base de datos.

Este tipo de bases de datos de diferencian de las bases de datos relacionales en tres puntos:

- **Jerárquico**: una base de datos XML establece una relación entre cada elemento en forma de jerarquía. En cambio, en el modelo relacional se establecen las relaciones mediante las tablas a través de un identificador.
- **Autodescriptivo**: los datos de un fichero XML son descriptivos, por tanto, una base de datos de este tipo también tendrá esta característica. Un fichero XML contiene información descriptiva y también sus datos, mientras que en las bases de datos relacionales se instauran diferentes tablas estructuradas con columnas y son más complejas de interpretar si se comparan con un XML.
- **Inherencia**: una base de datos XML seguirá un orden estricto según aparezcan los datos en el documento. Por su parte, en la base de datos relacional, los datos aparecerán sin orden a no ser que se especifique en la consulta qué columna de la tabla ordenar con un ORDER BY.

Ventajas e inconvenientes

Las ventajas que podemos destacar de una base de datos XML nativa son:

- Facilitan el acceso y el almacenaje de datos en formato XML sin necesidad de mapear la estructura de datos.
- Este tipo de base de datos incorpora un motor de búsqueda de datos muy potente.
- La facilidad con que se pueden añadir documentos a la base de datos.
- Permite guardar datos de forma heterogénea.
- Permite conservar la integridad de los datos y recuperarlos en su estado original.

Por otro lado, podemos citar algunos inconvenientes:

- Espacio: este tipo de base de datos ocupa una gran cantidad de espacio para guardar el mismo documento XML como formato de representación de la

información: al final se guarda todo un documento XML, las etiquetas que lo forman incluidas, y eso ocupa una gran cantidad de espacio.

- Limitación: es un tipo de base de datos que es limitado a solamente el uso de almacenamiento y búsqueda de ficheros XML.
- Dificultad para tratar los datos: estas bases de datos guardan la información en formato XML y es muy difícil generar nuevas estructuras de datos con esa información, por ejemplo, generación de estadísticas.
- Dificultades de indexación: es muy difícil indexar el contenido de una base de datos que ha de permitir la reducción del tiempo necesario para encontrar ciertos elementos clave.
- Modificación de los datos: esta base de datos limita la capacidad de modificación de los datos almacenados, ya que se tiene que sustituir todo el documento si se tiene que modificar un dato de dentro del fichero.

5.1.1. Gestores comerciales y libres. Instalación y configuración del gestor de base de datos XML

Ahora que conocemos mejor qué es una base de datos XML, vamos a introducir unos cuantos ejemplos para que podamos elegir el que más se adecúe a nuestra aplicación y sus propósitos. Podemos clasificarlos en bases de datos comerciales o libres. En la primera categoría podemos encontrar diferentes opciones, las más destacables son:

- **Excelon XIS**: es una base de datos XML de tipo comercial que ofrece una administración de los datos dinámica, extensible y muy fiable.
- **TAMINO**: se trata de una base de datos XML de un alto rendimiento y de gran disponibilidad. Es una base de datos XML nativa creada por la empresa SoftwareAG.
- **X-hive DB**: es otra opción de base de datos XML comercial que ofrece funciones de procesamiento avanzado y almacenamiento de datos en formato XML.

Por otro lado, también podemos encontrar diferentes opciones de código libre:

- **Oracle DB XML**: es otra base de datos de código abierto, es de tipo embebida y utiliza XQuery como lenguaje de consulta. Está basada en la base de datos Oracle db Berkeley y permite el manejo de transacciones y replicación.
- **eXist-db**: es una base de datos de código abierto codificado en Java que utiliza como lenguaje de consulta W3C Xquery. Con su instalación provee al usuario de un servidor para consumir peticiones REST.

- **Apache Xindice**: es una base de datos enfocada al almacenamiento de datos vía XML, de código abierto. La característica principal de esta base de datos es que no será necesario mapear el XML para otras estructuras de datos.

Instalación de la base de datos

La base de datos que usaremos para gestionar y almacenar información es la base de datos XML eXist-db. Esta base de datos se centra en el almacenamiento de documentos sin depender de un modelo relacional.

A continuación, os mostraremos cómo realizar la instalación. Este tipo de base de datos está especialmente diseñada para que tenga el mínimo de complicación durante la instalación. Antes de realizar la instalación, debemos verificar que en nuestro ordenador tenemos una versión de Java 1.8 o superior instalada. Para realizar la instalación, tenemos que descargarnos la última versión de la base de datos, a través de este enlace:

```
http://exist-db.org/exist/apps/homepage/index.html
```

En la web veremos que se nos descarga un archivo .jar. Este archivo no se instala de la misma forma que un archivo .exe. Para poder realizar la instalación, en primer lugar, iremos al directorio raíz de nuestro ordenador (normalmente es la C:) y allí crearemos una carpeta que llamaremos "eXist-db", donde copiaremos el .jar que nos hemos descargado de la página web. Después tendremos que abrir la consola de comandos de Windows. Para los que no sepan abrir la consola, solo es necesario ir al icono de Windows y buscar símbolo de sistema o escribir CMD. Una vez abierta, tendremos que dirigirnos a la carpeta que hemos creado y que es donde se encuentra el programa que queremos instalar. Para ello, ejecutaremos este comando:

```
cd c:\exist-db
```

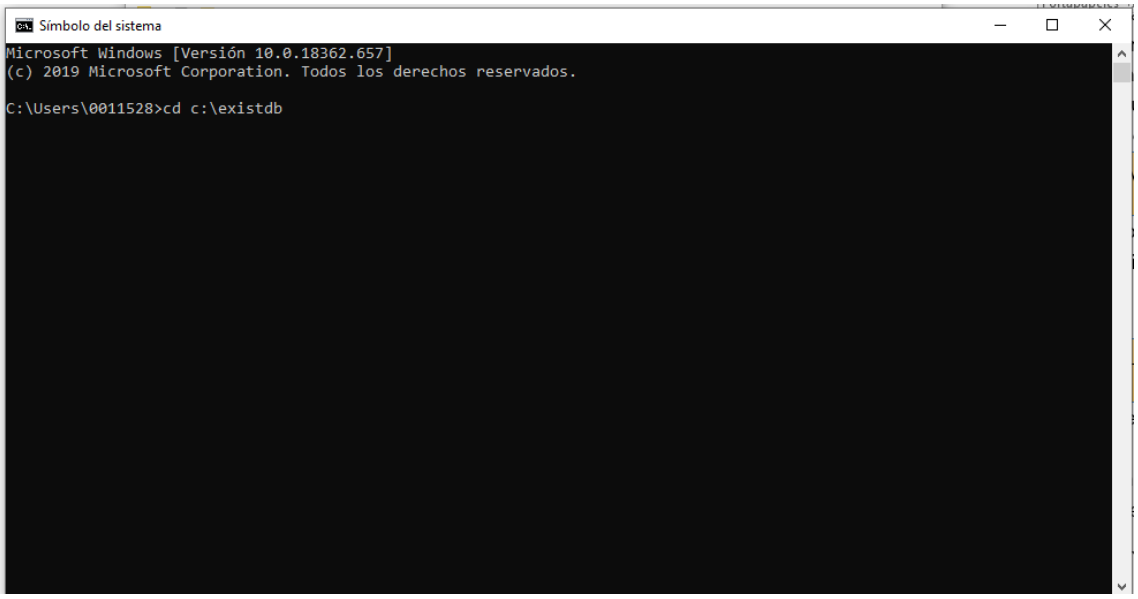


Ilustración 50. Consola de comandos de Windows con los comandos para acceder a la carpeta eXist-db.

Este comando sirve para dirigirnos a la carpeta "eXist-db", si habéis creado en el disco otra carpeta con otro nombre, deberéis poner esa. Siempre usaremos la que hemos creado y donde se encuentra el fichero .jar.

```
java –jar nombre_fichero_descargado.jar
```

Tendremos que ejecutar este comando, teniendo en cuenta que tendremos que añadir el nombre de nuestro .jar. El nombre debe ser parecido al que os mostramos en el cuadro de arriba. Estos comandos nos van a permitir definir los parámetros de configuración de la base de datos en nuestro sistema.

El primer paso será elegir un directorio para la instalación. Debemos tener en cuenta que tenemos que definir uno nosotros y evitar las carpetas por defecto “Program Files” o “Archivos de programa”, pues este tipo de directorios pueden ocasionarnos problemas en un futuro con los permisos de Windows. Por eso, es recomendable crear una carpeta "eXist-db" en el disco C: de nuestro ordenador, tal y como hemos comentado anteriormente.

Al ejecutar los comandos anteriores, tiene que abrirse una ventana parecida a esta, y pulsáis sobre *Sí*.

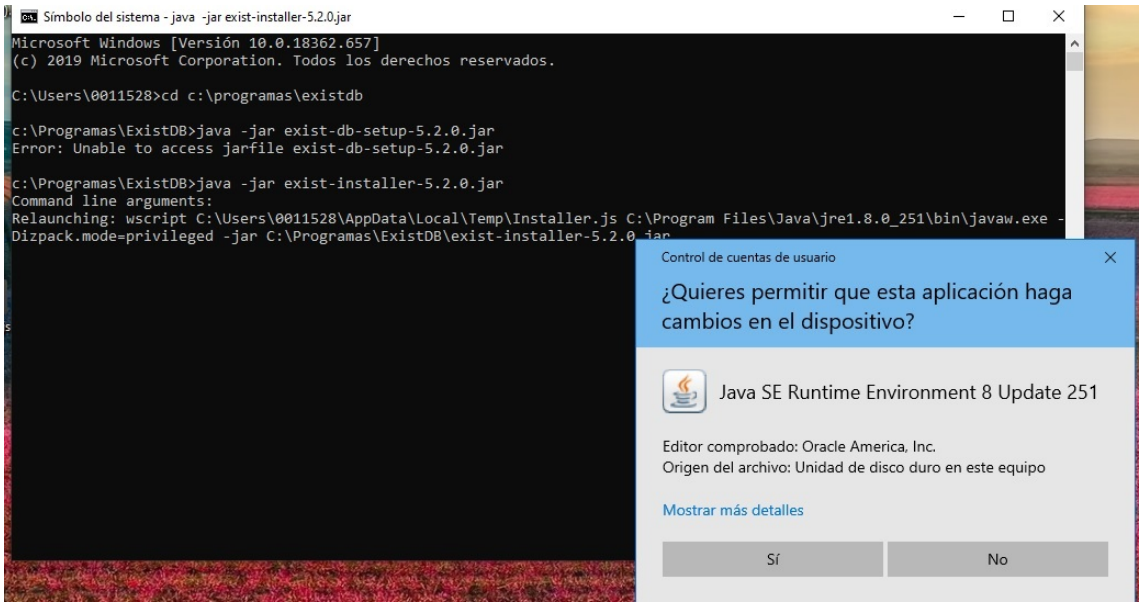


Ilustración 51. Captura con la pantalla que se ejecuta al escribir los comandos.

Deberá abrirse una ventana del instalador de la base de datos, como esta captura:

Ilustración 52. Captura con el instalador de eXist-db.

Cuando nos salga esta ventana, solo tenemos que pulsar sobre *Next*.

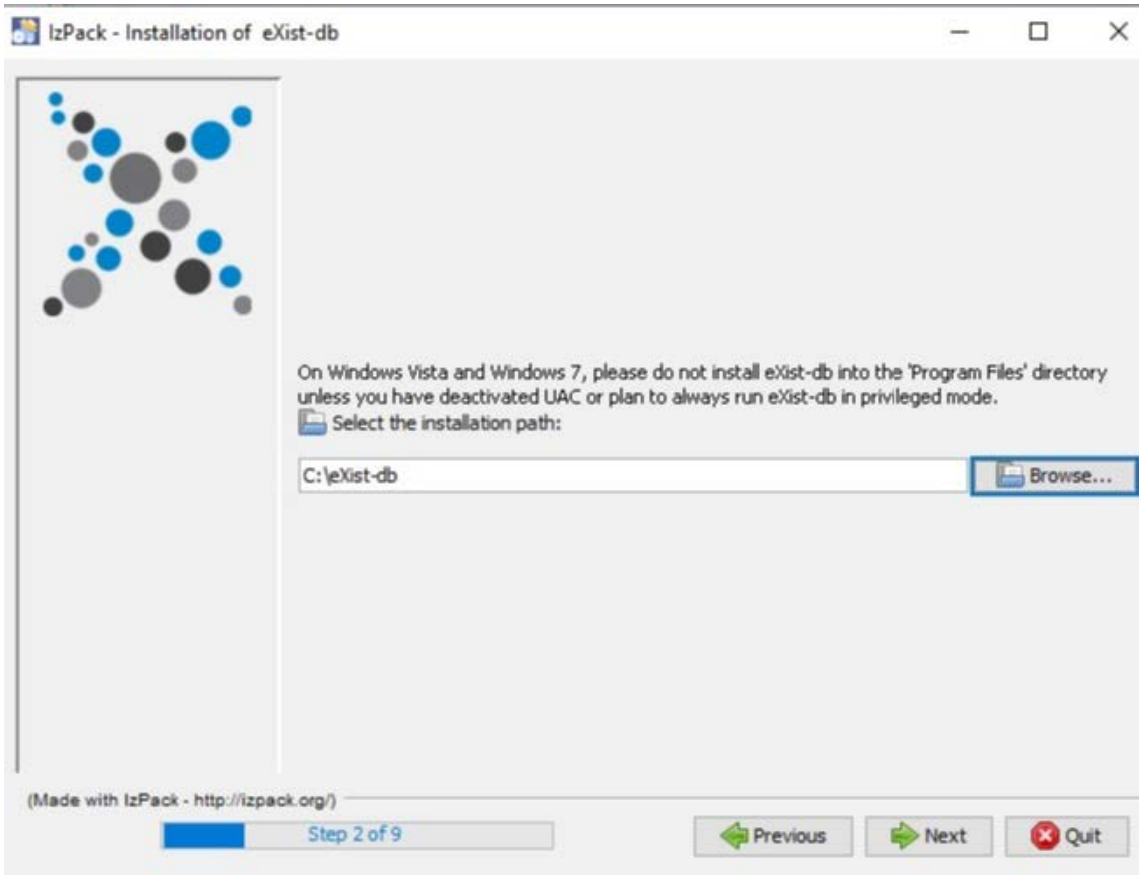


Ilustración 53. Pantalla con la ruta que debemos añadir para la instalación.

En este paso tenemos que elegir la carpeta que hemos creado anteriormente tal y como se muestra en pantalla. Esto será importante para poder trabajar correctamente con la base de datos y no tener problemas de permisos.

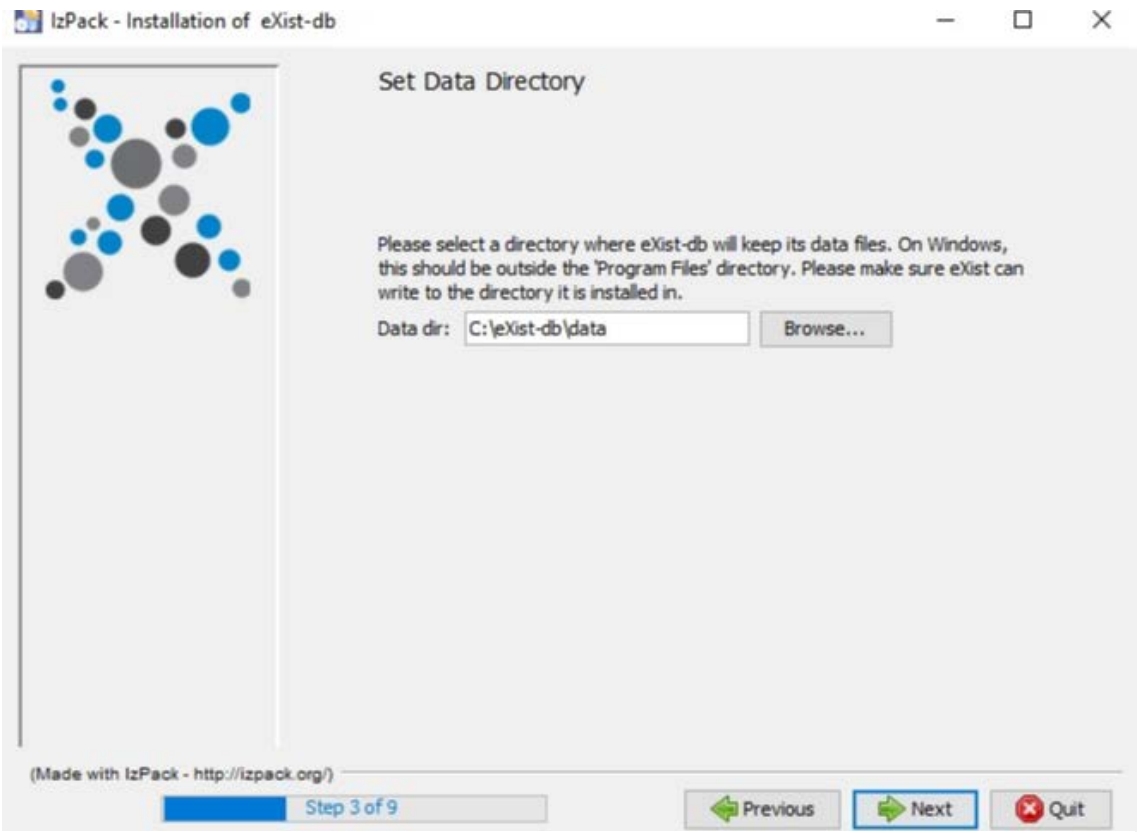


Ilustración 54. En esta pantalla solo tendremos que pulsar sobre Next.

En este paso, tendremos que definir la contraseña de acceso a la base de datos.

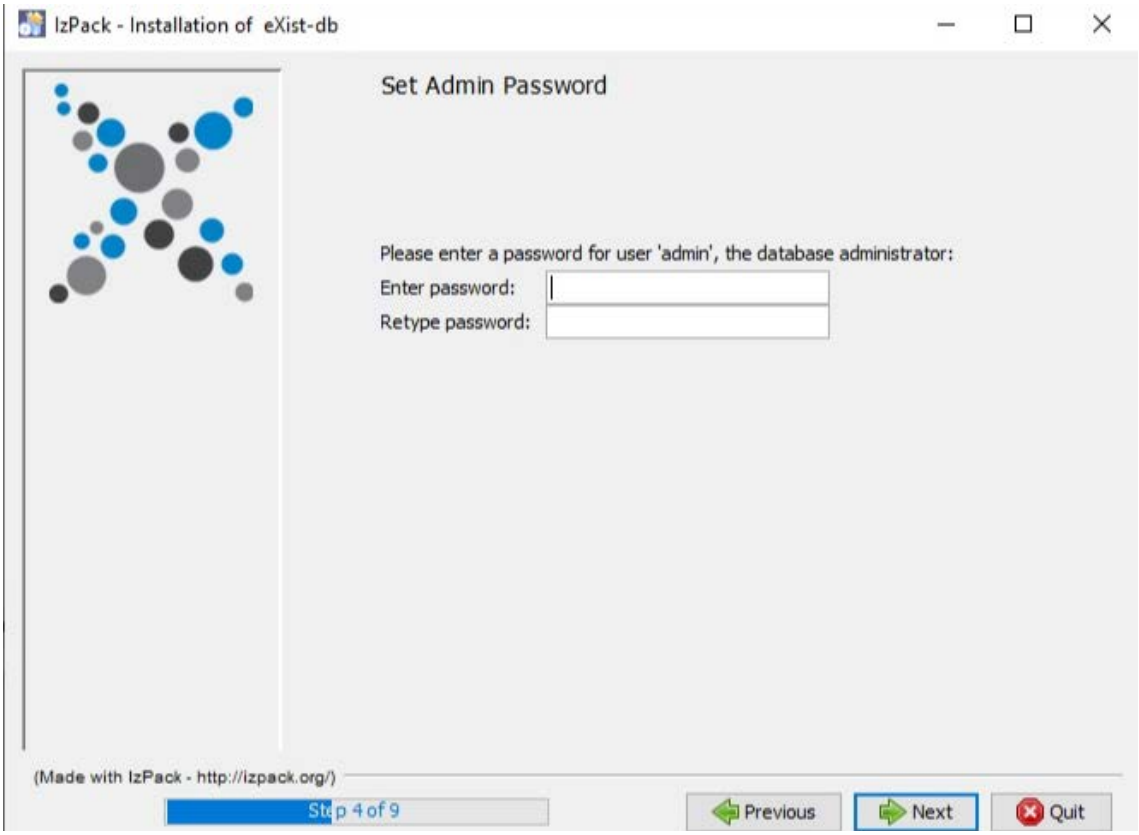


Ilustración 55. Paso en el que definiremos una contraseña para la base de datos.

En los siguientes pasos, solo tendremos que ir dándole al botón *Next* hasta que la instalación finalice del todo. Una vez finalizada, tendrá que aparecer esta pantalla:

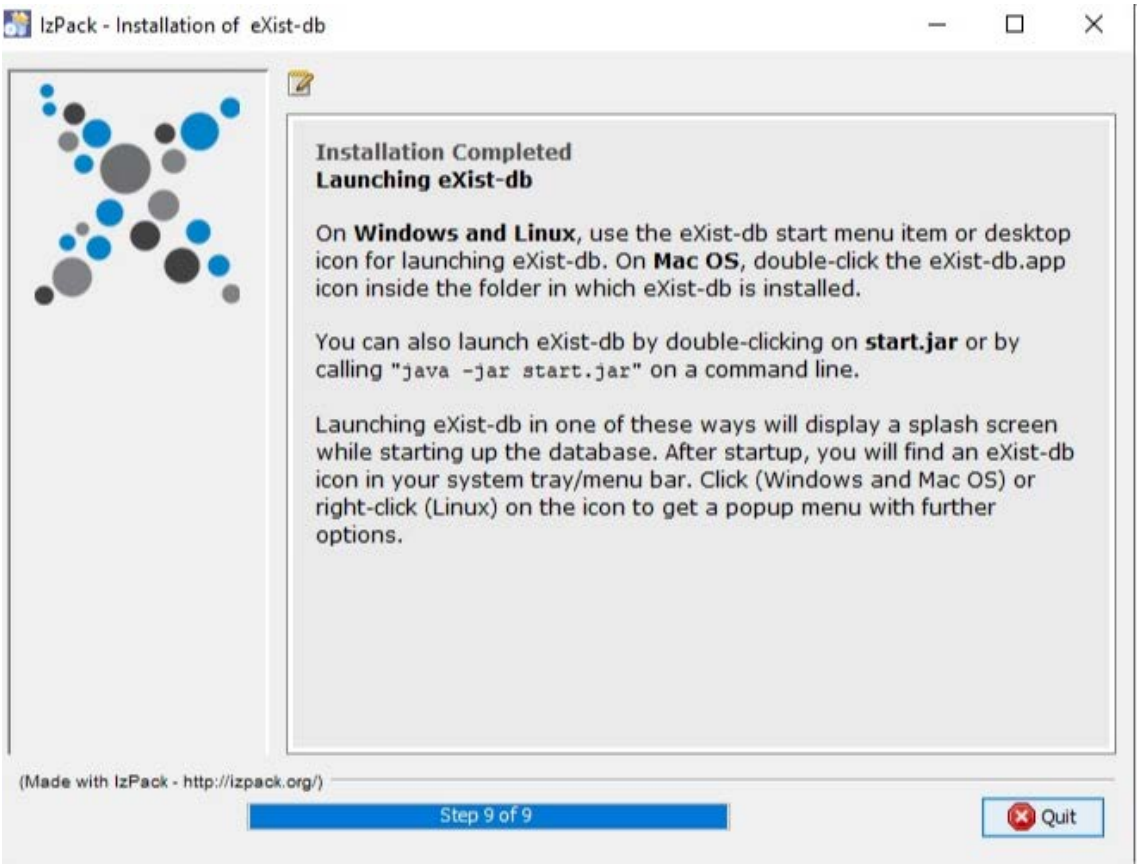
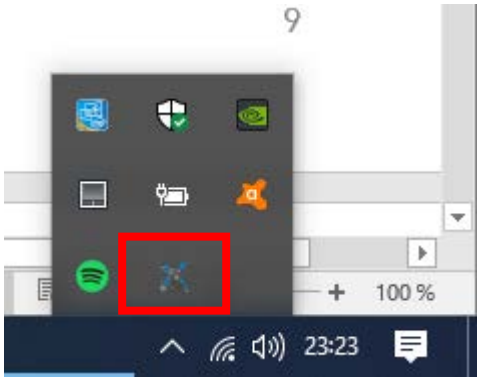


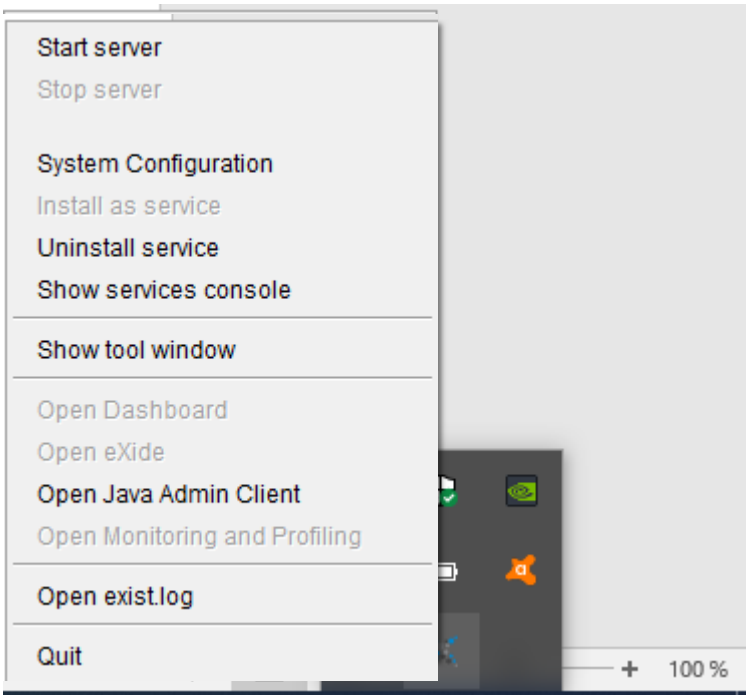
Ilustración 56. Finalización de la instalación.

Esta base de datos utiliza por defecto los puertos 8080 para http y el 8443 para https (sobre todo para interfaces SSL), por lo que tenemos que tener en cuenta que, si vamos a utilizar esta base de datos, tenemos que revisar que nada más utilice estos dos puertos o nos encontraremos con conflictos entre las dos aplicaciones.

Si queremos inicializar el servicio de la base de datos tendremos que abrir el programa. Una vez inicializado, la primera vez nos pedirá instalar un servicio mediante una ventana emergente, le indicamos que sí. Cuando esté abierta la base de datos, nos aparecerá un icono con una X como la de la captura:



Tendremos que abrir el menú de la base de datos clicando encima de la X con el botón derecho del ratón y se abrirá un menú con opciones:



Para que funcione la base de datos, tenemos que inicializar el servidor interno de la base de datos. Esto permitirá acceder a ella a través de su aplicación, pero también a través del navegador con esta URL:

```
http://localhost:8080/exist/apps/dashboard/index.html
```

5.2. Estrategias de almacenamiento

¿Qué podemos entender por estrategia de mapeo?

Las **estrategias de mapeo** son las diferentes opciones para almacenar datos en una base de datos XML, según la estructura del documento.

El objetivo principal de cada estrategia es definir un método para poder mover datos entre un documento XML y la base de datos correspondiente. Podemos encontrar diferentes tipos de estrategias, ya sean más simples o más complejas. Vamos a clasificar las existentes en tres métodos básicos:

1. **Almacenamiento como *BLOB*:** este tipo de almacenamiento se realiza en bases de datos relacionales. La columna que guardará nuestro documento XML será de tipo *BLOB*. Esta estrategia es especialmente útil cuando el archivo contiene contenido estático y solo se modificará para reemplazar el contenido de la columna por otro nuevo. Este tipo de estrategia es muy habitual para datos centrados en el documento a medida que se obtienen los datos y se almacenan como en una unidad. Es fácil de implementar ya que no requiere mapeo, pero a cambio, y como consecuencia, provoca una limitación de la búsqueda y la indexación del documento. El documento XML no se modifica una vez que es guardado en la base de datos, por lo tanto, minimiza el esfuerzo de recuperación.
2. **Almacenar versión modificada**: este tipo de estrategia se caracteriza por guardar una modificación del documento completo en el sistema de archivos. Esta estrategia es especialmente útil cuando tenemos un número reducido de documentos y los XML no se actualizan a menudo ni se transfieren a otros sistemas. Si utilizamos este método, debemos ser consciente de sus limitaciones, como la escalabilidad, la flexibilidad en el almacenamiento o la recuperación de los datos, y una seguridad baja. Puede ser una opción beneficiosa si no tenemos una gran cantidad de documentos que guardar.
3. **Mapeo de los datos**: esta estrategia se caracteriza por realizar un mapeo de los datos del documento en la base de datos, siempre que el documento XML no siga la misma estructura que la base de datos, es decir, los documentos que no sean compatibles tendrán que ser mapeados para que se puedan transformar y adaptar a la base de datos.

5.3. Establecimiento y cierre de conexiones

Para establecer la conexión tendremos que usar la API **XML:DB API**, especialmente útil para la construcción de una aplicación con una base de datos XML nativa.

Los componentes que se utilizan principalmente con esta API son controladores, colecciones, recursos y servicios.

Los controladores son implementaciones de la interfaz de la base de datos que encapsulan la lógica de acceso a la base de datos para productos de bases de datos XML específicos. Los proporciona el proveedor del producto y deben registrarse con el administrador de la base de datos.

Una colección es como un directorio que ayuda a establecer una jerarquía y organización para almacenar los recursos o subcolecciones. Actualmente, la API define dos recursos diferentes: XMLResource y BinaryResource. Un XMLResource representa un documento

XML o un fragmento de documento seleccionado por una consulta XPath ejecutada previamente. Por otro lado, un BinaryResource, representa un documento con datos binarios.

Por último, un servicio se utiliza para tareas como consultas de colecciones a través de XPath o para administrar colecciones.

Para establecer las conexiones, podemos destacar estas interfaces:

- ***DatabaseManager**:* es una clase que nos va a permitir el acceso a la base de datos
- ***Collection**:* esta clase se encarga de representar una colección, un conjunto de recursos de una base de datos XML.
- ***XMLResource**:* se trata de una clase que proporciona acceso a los recursos XML almacenados en una base de datos.
- ***CollectionManagementService**:* esta interfaz permite la gestión básica de las colecciones de una base de datos: crear y borrar colecciones.
- ***XPathQueryservice**:* esta interfaz permite la ejecución de consultas XUpdate en el contexto de una colección completa o de un solo recurso XML.

Para establecer la conexión, como muchas aplicaciones, es necesario añadir una librería a nuestro proyecto. Para ello tenemos dos opciones: añadiendo las dependencias en el fichero pom.xml o añadiendo la librería directamente a las dependencias del proyecto. Las librerías que necesitaremos son:

- Si usamos un proyecto Maven, añadiremos estas dos dependencias en el fichero pom.xml.

```
<dependency>
    <groupId>net.sf.xmldb-org</groupId>
    <artifactId>xmldb-api</artifactId>
    <version>1.7.0</version>
</dependency>

<dependency>
    <groupId>org.exist-db</groupId>
    <artifactId>exist-core</artifactId>
    <version>5.2.0</version>
    <scope>runtime</scope>
</dependency>
```

La versión puede variar de la que mostramos en el ejemplo, es recomendable usar la última versión siempre. Si queremos verificar que la que usamos es la más nueva, podemos consultar la web del repositorio de Maven:

https://mvnrepository.com/

- Si usamos un proyecto Java, descargaremos estas dos librerías: org-exist-db y net.sf.xmldb-org.

> https://repo1.maven.org/maven2/net/sf/xmldb-org/xmldb-api/1.7.0/xmldb-api-1.7.0.jar
>
> https://repo1.maven.org/maven2/org/exist-db/exist-core/5.2.0/exist-core-5.2.0.jar

A continuación, para establecer la conexión entre la base de datos y nuestra aplicación, necesitamos crear una clase que sea la auxiliar que realice esta tarea. Como hemos visto en temas anteriores, es una tarea bastante común en muchas otras bases de datos de otros tipos, y en este tipo de base de datos se realiza del mismo modo.

En primer lugar, tendremos que registrar el *driver*. Para ello, usaremos el *driver* de la base de datos eXist-db: org.exist.xmldb.DatabaseImpl.

```
Class cl = Class.forName("org.exist.xmldb.DatabaseImpl");
Database database = (Database) cl.newInstance();
database.setProperty("create-database", "true");
DatabaseManager.registerDatabase(database);
```

Como vemos en el recuadro, el método de registro del *driver* para esta base de datos es algo más extenso que en otras bases de datos. En la primera línea, definimos una nueva instancia de la clase *Class*, la cual inicializaremos realizando una llamada al método de este objeto *forName* pasándole por parámetro el *driver* mencionado anteriormente. A continuación, tenemos que definir una nueva instancia de *Database*, la cual inicializaremos a través de la clase *Class* y haciendo una llamada al método *newInstance()*.

El siguiente paso será indicar a la base de datos que necesitamos crear una nueva instancia. Para ello, utilizaremos la clase *Database* y llamaremos al método *setProperty()*. Por parámetro indicaremos dos valores: primero, que necesitamos crear una base de datos con *create-database;* y, en segundo lugar, *true* para indicarle que se hace el registro a la base de datos. Finalmente, con *DatabaseManager* y el método *registerDatabase* le pasaremos el objeto *database* para finalizar el registro. Este paso solo tenemos que realizarlo una vez al establecer la conexión.

A continuación, obtenemos un objeto *Collection* del *DatabaseManager* usando la llamada del método estático *DatabaseManager.getCollection()*. Este método espera que pasemos por parámetro un URI totalmente válido que será la dirección de acceso a la base de datos. El formato de este URI debe ser:

```
xmldb:exist://localhost:8080/nombre_bbdd/coleccion
```

La primera parte del URI es *xmldb:exist*, que especificará la base de datos que estamos usando. En este caso, es una base de datos XML y se trata de la base de datos eXist. Como podemos registrar más de un controlador de base de datos con el administrador de base de datos, es necesario establecer esta primera parte obligatoriamente. El administrador identifica esta primera parte para seleccionar el controlador adecuado. Seguidamente, añadimos el servidor que vamos a utilizar: al realizar las pruebas en nuestro ordenador, se utilizará el término "localhost" (que es el que indica que utilizamos un *host* local) y seguidamente el puerto HTTP.

La parte final del URI identifica la ruta de recopilación y, opcionalmente, la dirección de *host* del servidor de la base de datos en la red. Internamente, eXist usa dos implementaciones de controladores diferentes: la primera se comunica con un motor de base de datos remoto utilizando llamadas XML-RPC; la segunda tiene acceso directo a una instancia local de eXist-db. La colección raíz siempre se identifica con */db*.

5.4. Colecciones y documentos. Clases para su tratamiento

En este apartado, haremos una introducción al concepto de colecciones y documentos en la base de datos eXist-db y también qué clases podemos utilizar para su tratamiento. La base de datos soporta el concepto de colecciones de documentos.

5.4.1. Colecciones

Las colecciones son grupos de documentos XML, documentos binarios y otras colecciones. Son el equivalente a las carpetas de nuestro sistema operativo.

En resumen, una colección reúne un numero de documentos XML que se organizan siguiendo una estructura jerárquica similar a las carpetas de un sistema operativo. Una

colección agrupará un conjunto de documentos XML o archivos binarios. La colección tendrá un nombre y se identificará mediante un URI. Cada colección guardará sus propios metadatos, como también los metadatos de los documentos que se encuentran dentro de esa colección.

Dentro de las colecciones, podemos encontrar lo que se denomina un recurso, que sería el equivalente a un fichero o un documento. Para poder realizar acciones con estas colecciones, haremos uso de los servicios, que nos van a permitir hacer búsquedas o insertar información a nuestro documento XML.

Como veíamos en el apartado anterior, el uso de conexiones es algo usual para acceder a la información de la base de datos y se realiza mediante un URI. Para acceder a las colecciones, se utiliza el URI que hemos comentado en el apartado anterior sumándole */db*, que será la indicación que usa eXist-db para indicar que a partir de ese punto serán colecciones. */db* es la raíz del sistema de archivos UNIX. A partir de ahí, podemos crear subcolecciones en */db* para almacenar datos.

```
/db   <- Colección raíz
/db/ilerna/      <- Subcolección
/db/ilerna/productos
```

Las colecciones contienen dos tipos de datos: archivos o subcolecciones. Los archivos en eXist se denominan recursos. Los documentos XML bien formados son indexados por eXist. Los archivos no XML se almacenan como objetos binarios.

Para mantener una estructura lógica en la base de datos, se recomienda la creación de subcolecciones. Las subcolecciones serían las colecciones que creamos dentro de otra colección. Será especialmente útil para mantener un orden. Por ejemplo, si tenemos una colección llamada "productos", podremos crear diferentes subcolecciones para guardar los diferentes tipos de productos. Por tanto, una subcolección de productos podría ser alimentos, bebidas o productos de limpieza, por ejemplo. Dentro de esta subcolección, podríamos añadir nuestros documentos.

Esta estructura nos ayudará en la organización de los datos y a seguir una estructura lógica. Si no seguimos una estructura, la base de datos estará mal organizada y la consulta de los datos será mucho más difícil, por eso se recomienda la creación de subcolecciones y que estas no tengan más de 1.000 archivos. Esta es una pauta no obligatoria, pero sería una buena práctica que seguir.

La base de datos eXist puede consultar todos los documentos XML en cualquier colección o subcolección para encontrar un documento almacenado. A diferencia de un sistema de archivos, no necesita saber en qué año o mes están los datos, solo necesita conocer la colección raíz para iniciar sus consultas: eXist encontrará sus documentos desde allí.

5.4.2. Documentos

Los documentos son una unidad básica con la que trabaja eXist, y podemos dividirlos en dos clases: los documentos XML y los documentos binarios. En esta base de datos podemos organizar la información de los documentos como nos parezca, ya que eXist no tiene una norma para organizar la información, ya sea en un único archivo grande o en muchos en formato más reducido.

Sin embargo, la organización de los datos es importante, por lo que deberíamos seguir estas buenas practicas:

- Documentos más pequeños: si tenemos almacenados algunos documentos de tamaño grande y se intenta acceder a ellos de manera simultánea, se pueden ocasionar conflictos. En cambio, si tenemos muchos documentos pequeños y muchas peticiones de consulta, se pueden realizar de forma independiente, porque los datos están más repartidos. Del mismo modo, cuando estamos actualizando un documento, se bloquea la escritura. El acceso a los datos a través de la aplicación también bloquea la escritura de los datos.
- Lectura: cuando se realiza una petición de lectura de datos, ese fichero queda bloqueado hasta que se ha terminado la lectura, por eso es recomendable tener ficheros más pequeños, porque, si tenemos un fichero muy grande con mucha información, será más lento tener acceso a él si hay muchas peticiones de lectura o actualización de los datos.

5.4.3. Clases para su tratamiento

Tal y como hemos explicado en el apartado 5.3, para realizar una aplicación Java, es necesaria la API XML DB. Esta interfaz nos va a proporcionar una serie de clases que nos permitirán gestionar tanto las colecciones como los ficheros.

Clase *Collection*

Esta clase representa una colección de recursos almacenados en un documento XML en la base de datos.

De esta interfaz podemos destacar los métodos más importantes y que nos serán más útiles a lo largo de la construcción de nuestra aplicación:

Tipo	Método	Descripción
void	close()	Este método se encarga de cerrar todos los recursos abiertos de la interfaz *Collection*.
String	createdId()	Se encarga de crear un id nuevo y único dentro del contexto de la clase *Collection*.
Resource	createResource(String id, String tipo)	Este método se encarga de crear una nueva instancia vacía de la clase *Resource* con el id que se proporciona por parámetro.
Collection	getChildCollection(String nombre)	Busca si existe una colección hija que le indicaremos por parámetro y devuelve una nueva instancia de *Collection* si existe.
int	getChildCollectionCount()	Devuelve el número de colecciones hija que existen o 0 si no existe ninguna.
String	getName()	Devuelve el nombre de la instancia de la colección en un *string*.
Collection	getParentCollection()	Devuelve la colección padre de la colección que llama al método o nulo si no existe.
Resource	getResource(String id)	Se encarga de devolver una instancia de *Resource* con la información de la base de datos.
int	getResourceCount()	Devuelve el número de recursos almacenados en la colección o 0 si no encuentra ninguno.
Service	getService(String nombre, String version)	Este método devuelve una instancia del nombre y la versión del *Service* que le hemos pasado por parámetro.
Service[]	getServices()	Se encarga de devolver una lista de todos los servicios conocidos por la colección.

boolean	isOpen()	Se encarga de devolver *true* o *false* según la instancia de *Collection* esté abierta o no.
String[]	listChildCollections()	Este método devuelve una lista de nombres de colecciones que nombran todas las colecciones hijo de la colección actual.
String[]	listResources()	Se encarga de devolver una lista de los identificadores de los recursos almacenados en la colección.
void	removeResource(Resource res)	Se encarga de borrar el recurso que le indicamos por parámetro de la base de datos.
void	storeResource(Resource res)	Se encarga de guardar el recurso que le pasamos por parámetro a la base de datos.

Clase *Document*

Para la representación de documentos XML se utiliza la interfaz *Document*. Una instancia de la clase *Document* representará al elemento raíz de un fichero XML y permitirá el acceso principal a los datos que se encuentran en el documento.

Si pensamos en un documento XML, sabemos que está compuesto por nodos, elementos, atributos y valores, es por eso que esta clase proporcionará métodos que nos van a permitir obtener todos estos elementos de un documento XML.

Los métodos más importantes y destacables de esta interfaz son:

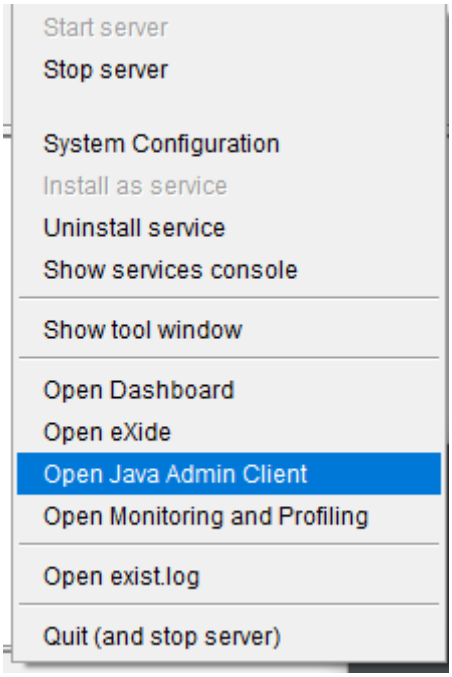
Tipo	Método	Descripción
Attr	**createAttribute**(String nombre)	Este método se encarga de crear un *Attr* (un atributo) a partir del nombre que le pasamos por parámetro.
Attr	**createAttributeNS(**String namespaceURI, String nombre)	Crea un atributo del nombre que le pasamos por parámetro y el URI de espacio de nombres.

Element	**createElement**(String tagName)	Crea un elemento del tipo que se le especifica por parámetro.
Element	**createElementNS**(String namespaceURI, String qualifiedName)	Crea un elemento del nombre que le pasamos por parámetro y el URI del espacio de nombres.
Text	**createTextNode**(String datos)	Crea un nodo de texto a partir del texto que le pasamos por parámetro.
Element	**getDocumentElement**()	Este método permite obtener el nodo secundario, que es el elemento del documento.
String	**getDocumentURI**()	Este método se encarga de obtener la URI del documento o devuelve nulo si no lo encuentra.
Element	**getElementById**(String id)	Este método se encarga de devolver el id del elemento que le pasamos por parámetro.
NodeList	**getElementsByTagName**(String nombreTag)	Devuelve una *NodeList* de todos los elementos en orden del documento a partir del nombre de etiqueta que le pasamos por parámetro y que se encuentren en el documento.
NodeList	**getElementsByTagNameNS**(String nombreURI, String nombreLocal)	Devuelve una *NodeList* de todos los elementos que coincidan con el nombre del URI y el nombre del espacio de nombres que le pasamos por parámetro y que se encuentren en el documento.
String	**getInputEncoding**()	Devuelve un atributo que especifica el *encoding* utilizado para este documento.
String	**getXmlEncoding**()	Este método especifica el *encoding* del XML que queramos obtener.
String	**getXmlVersion**()	Este método devuelve la versión del XML.
void	**setXmlVersion**(String version)	Este método establece el número de versión del XML.

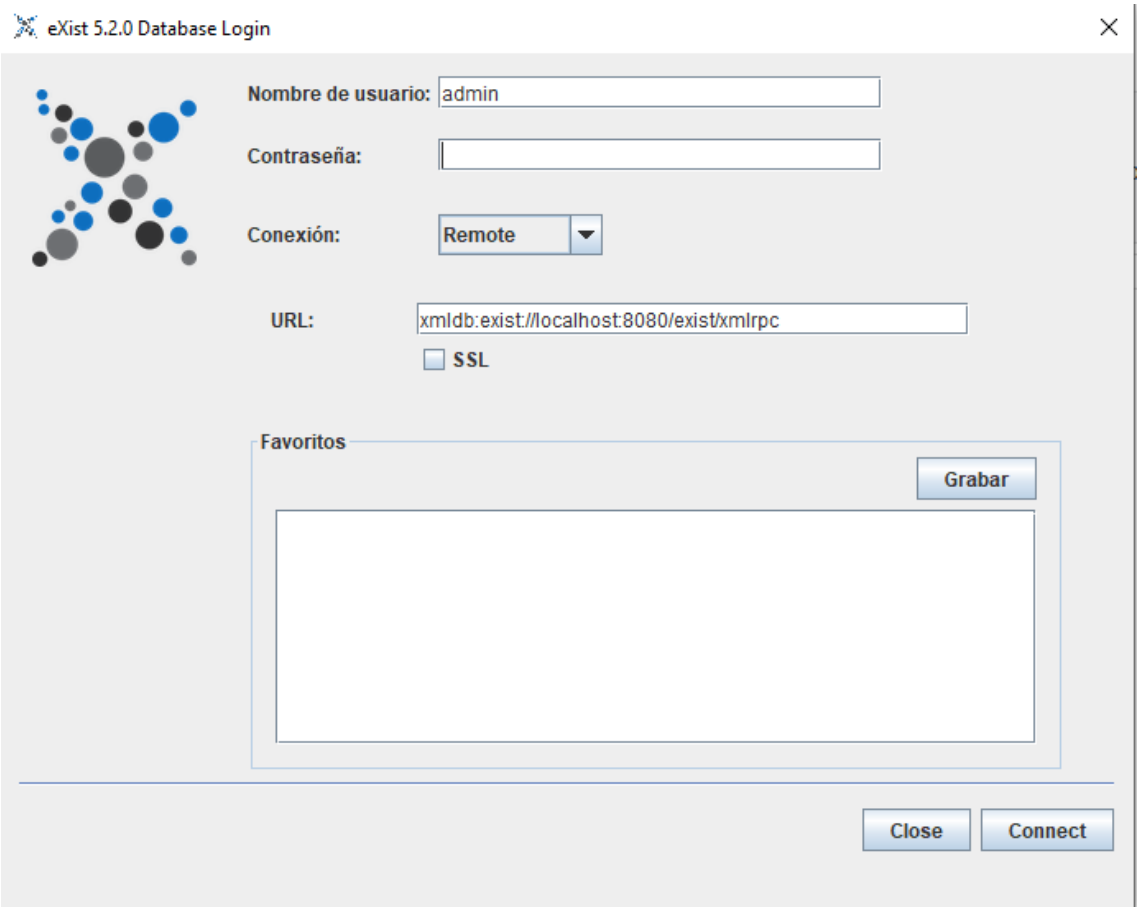
5.5. Creación y borrado de colecciones, clases y métodos

Cuando trabajemos con una aplicación que utiliza una base de datos del estilo de eXist-db, tendremos que almacenar la información en alguna estructura. Aquí es donde entran en juego las colecciones. En esta base de datos, las colecciones se crean bajo la colección de la base de datos *db/apps*. Si necesitamos crear una nueva colección será hija de la mencionada anteriormente. Para poder crear una colección, debemos asegurarnos que tenemos funcionando la base de datos, para lo que es importante revisar el apartado anterior.

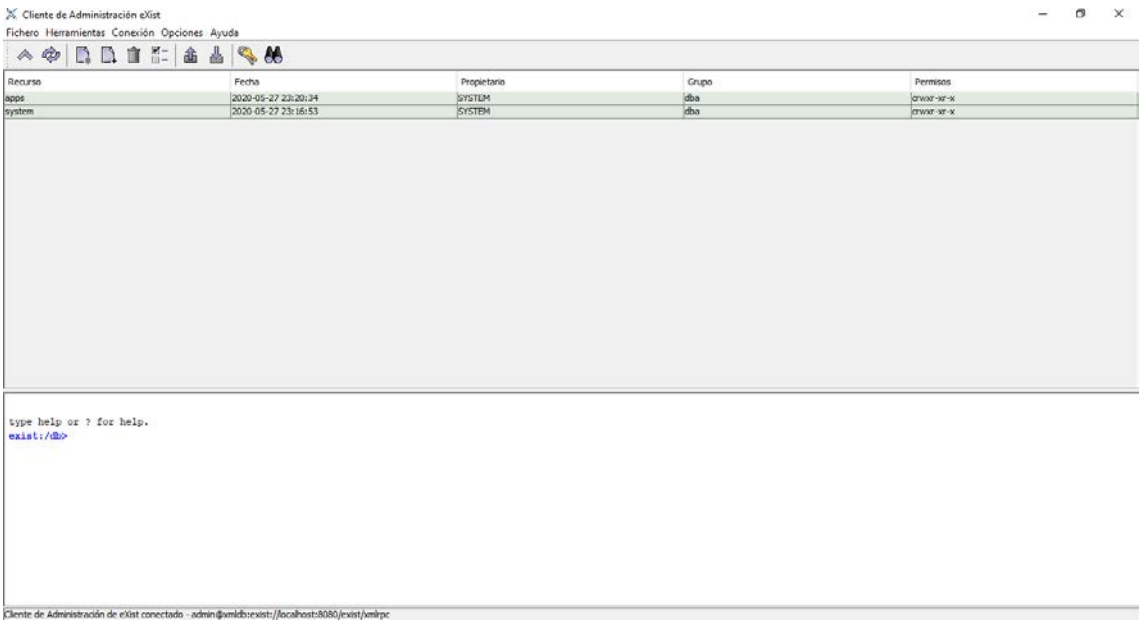
A través de la aplicación de escritorio de la base de datos, tendremos que crear las colecciones pulsando con el botón derecho encima del icono de la base de datos.



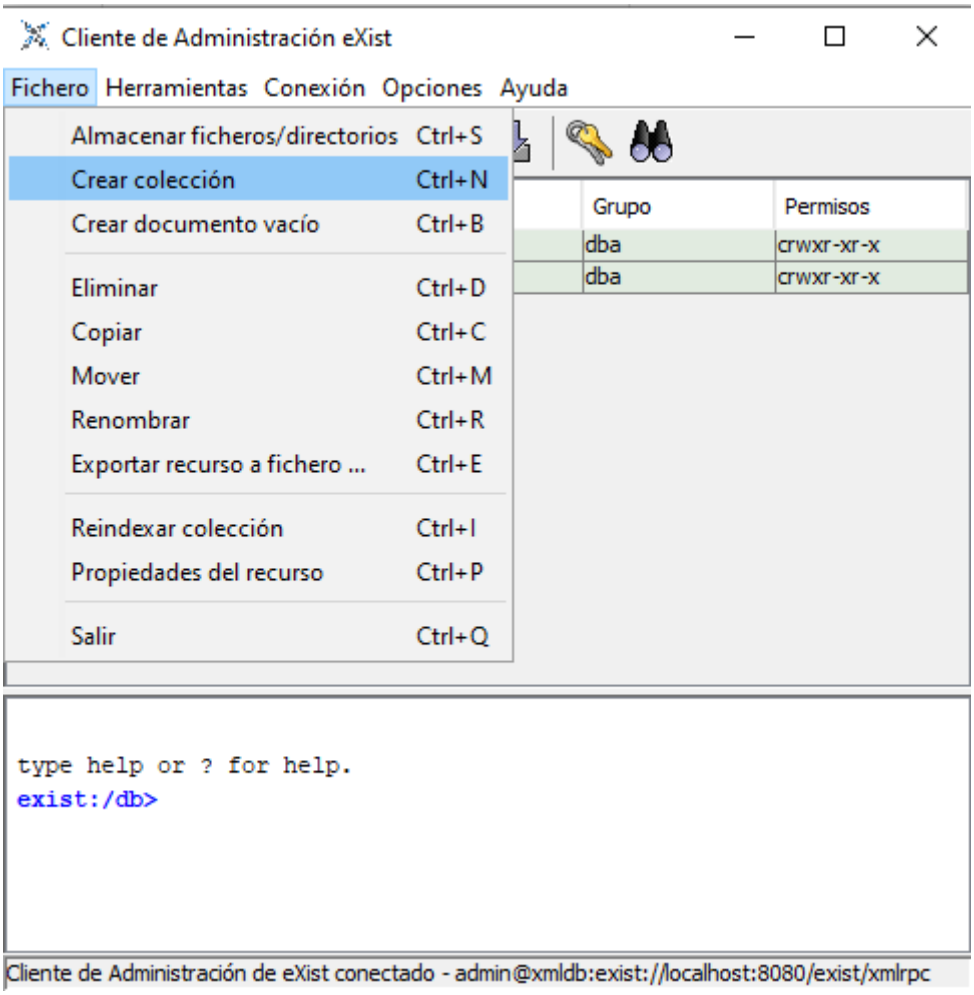
Seleccionaremos *Open Java Admin Client*, y se nos tendrá que abrir una ventana parecida a esta:



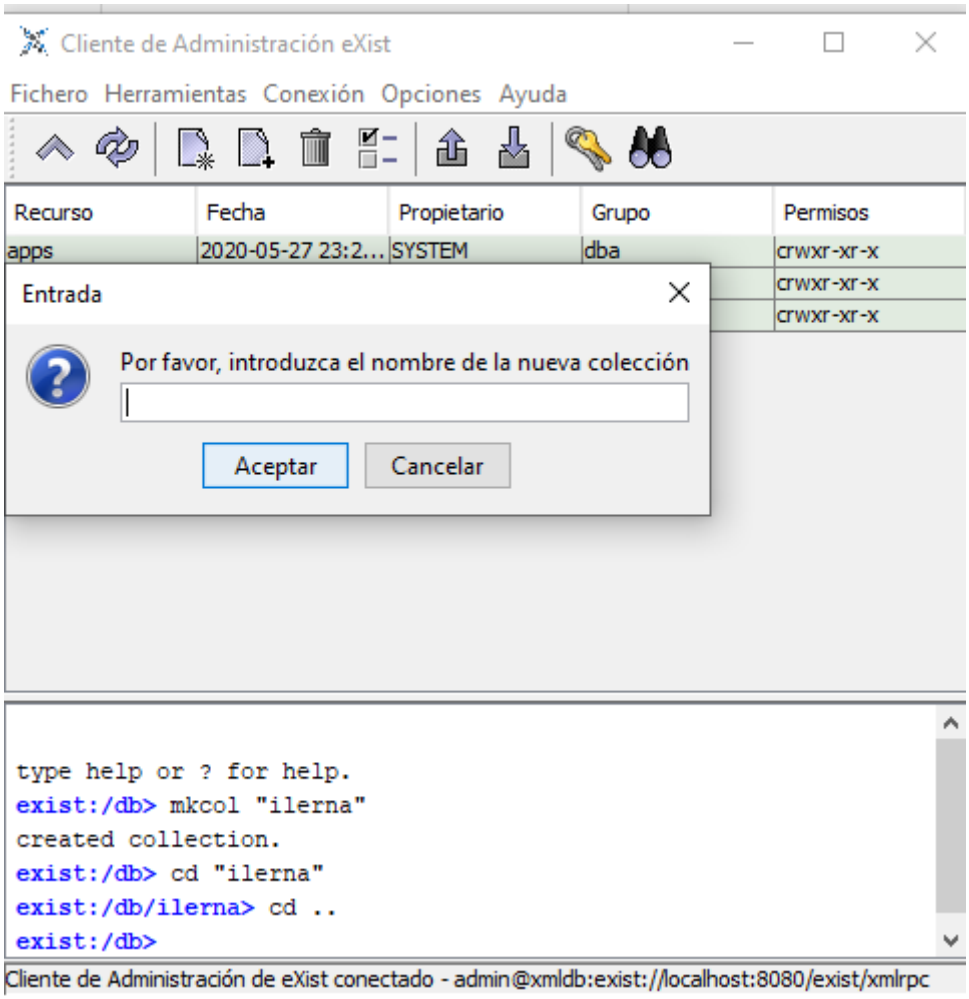
A continuación, tendremos que iniciar una sesión a nuestra base de datos. Para ello tendremos que clicar en el botón que encontraremos arriba a la derecha llamado *Login*. Tendremos que acceder a la base de datos bajo el usuario *admin* y con la contraseña que hemos guardado durante la instalación.



El siguiente paso será dirigirnos a la casilla llamada *Collections*, tal y como vemos en la captura.

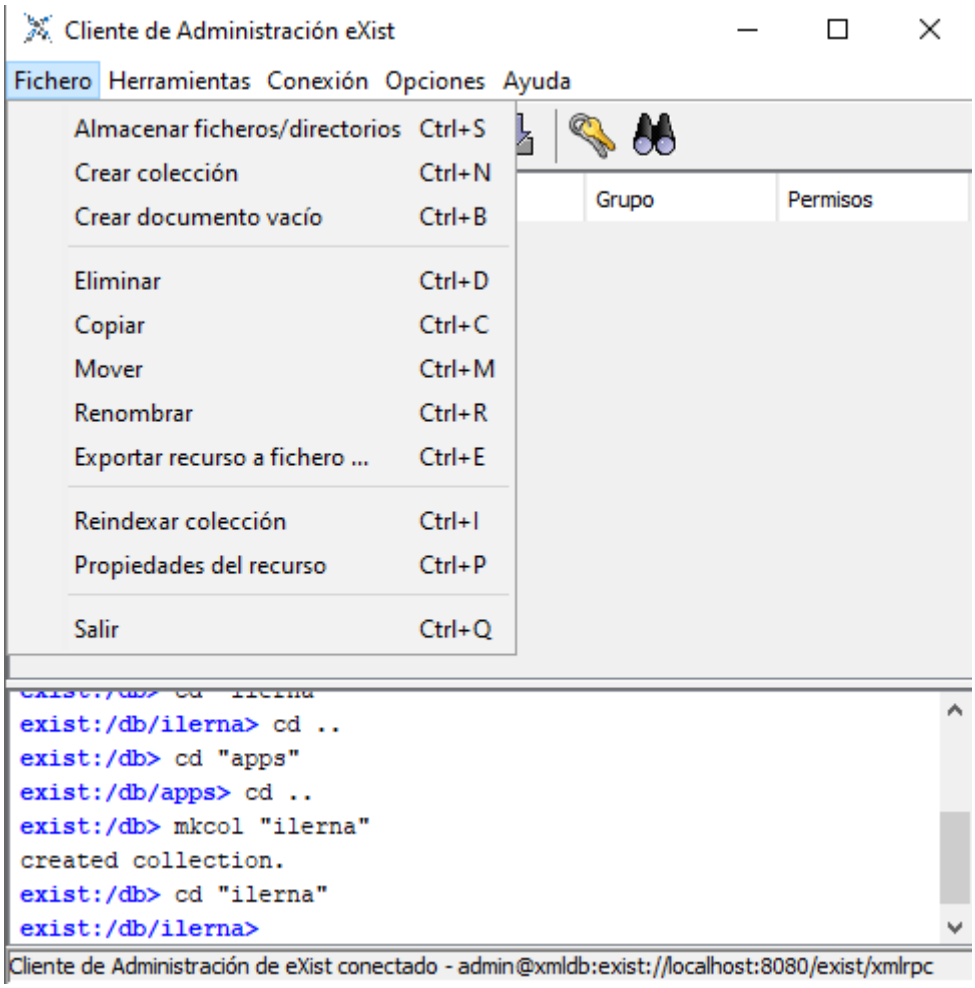


A continuación, tendremos que navegar a través de la colección principal *db/apps*. Cuando estemos dentro, podemos crear una nueva colección con el nombre que nos apetezca. Para este ejercicio, la llamaremos "ilerna". Lo tendremos que realizar dirigiéndonos a *Fichero > Crear colección* y se abrirá una ventana como esta:



Al darle a *Aceptar* ya tendremos creada una colección. Para crear una colección dentro de otra, solo tenemos que navegar a través de la colección, es decir, si queremos crear una colección nueva dentro de "ilerna", solo necesitaremos estar en la ruta: */db/apps/ilerna* y crear otra nueva dentro.

Para añadir archivos dentro de esta colección, tendremos que dirigirnos a la opción *Fichero > Almacenar ficheros/directorios* que encontraremos en el menú superior:



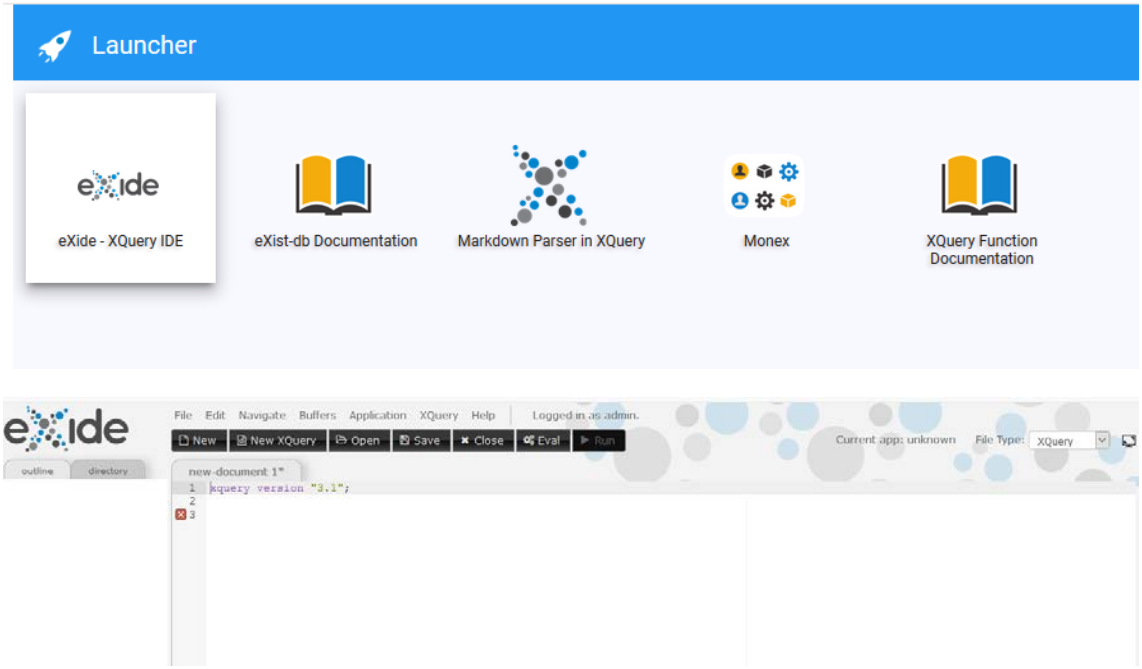
Para visualizar la información, imaginemos que en el paso anterior no hemos añadido ningún fichero en la base de datos. En este apartado os enseñaremos a visualizar el fichero XML o los recursos almacenados a través del navegador.

Para poder trabajar de manera más efectiva y versátil con los datos almacenados, tendremos que utilizar un editor de texto que eXist tiene vinculado. Este editor de texto se llama eXide y lo podremos utilizar para visualizar y editar ficheros. Los pasos que seguir son:

Tendremos que acceder a esta URL a través de nuestro navegador. Esta URL es la básica para acceder a la interfaz de la base de datos a través de un navegador, por eso será la que más utilizaremos para hacer nuestras gestiones:

```
http://localhost:8080/exist/
```

Tendremos que iniciar sesión otra vez si habíamos cerrado sesión. Seguidamente, tendremos que hacer clic sobre el título llamado *eXide-XQuery IDE*, lo que abrirá una ventana como esta:



Otra alternativa para visualizar un XML es a través de su URL. La ruta de un fichero XML dentro de la base de datos será a través de las colecciones más el nombre del fichero y su extensión.

Clases y métodos para la creación y borrado de colecciones

Para la gestión de las colecciones, la API XML DB nos ofrece la interfaz *Collection*. Como hemos comentado en el apartado 5.4, una colección es un directorio donde se encontrarán todos los documentos XML de la base de datos que queramos almacenar.

Aparte de *Collection*, haremos uso de ***CollectionManagementService***. Se trata de una clase que permite la gestión de las colecciones que se encuentran en bases de datos.

Los métodos necesarios para crear colecciones son:

Tipo	Método	Descripción
Collection	createCollection(String name)	Método que se encarga de crear una nueva colección en la base de datos.

void	removeCollection(String nombre)	Método que se encarga de borrar una colección que le indiquemos por parámetro.
void	removeCollection(string colección, string destinación, string nuevoNombre)	Este método se encarga de mover de sitio una colección.
void	moveResource(String resourcePath, String destinacionPath, String Nuevo nombre)	Se encarga de mover un recurso al *path* indicado.

Para ver cómo crear una colección, vamos a mostrar un ejemplo de implementación:

```
Collection coleccionPadre =
DatabaseManager.getCollection("xmldb:exist://localhost:8080/ilern
a/");
String nuevaColeccion = "ejemplo1";
Collection coleccion =
coleccionPadre.getChildCollection(nuevaColeccion);

if (coleccion == null) {

      CollectionManagementService mgmt =
      (CollectionManagementService)
      coleccionPadre.getService("CollectionManagementService",
      "1.0");

      coleccion = mgmt.createCollection(nuevaColeccion);

      // Cerramos la conexión
      coleccionPadre.close();

}
```

En primer lugar, tenemos que crear un *string* para definir el nombre de la colección que queremos crear. A continuación, con la ayuda de la clase *Collection*, crearemos una nueva instancia de esta clase. Antes tenemos que tener creada otra colección que indique la ruta en la que se creará la nueva colección. Con esa colección padre crearemos la nueva con un *getChildCollection* y asignaremos el nombre de la nueva colección. Ahora tenemos el objeto Java con la nueva colección, el siguiente paso será

crearla en la base de datos: para ello tenemos que tener una nueva instancia de la clase *CollectionManagementService* y rellenaremos esta clase con esta llamada a partir de la colección *current.getService("CollectionManagementService", "1.0");*. Lo que hace es llamar al servicio de la base de datos y setear la información a la nueva instancia creada. A continuación, usaremos el servicio llamando a *createCollection* y asignando la colección nueva que queremos crear, en este caso, hija. Con este paso, tendríamos creada la nueva colección en la base de datos. El control de excepciones se hará con *XMLDBException*, que es la clase que gestiona todos los errores que pueden ocurrir con esta API.

Para el borrado, la implementación será bastante parecida, pero, en lugar de utilizar el método *createCollection*, usaremos *removeCollection*. Vamos a visualizar un ejemplo de su implementación:

```
try{
	String name = "collection";
	CollectionManagementService mgmtService =
	(CollectionManagementServiceImpl)coll.getService("Collection
	ManagementService", "1.0");
	mgmtService.removeCollection(name);

}catch(XMLDBException e){
	e.getMessage();
}
```

En este caso, tenemos que crear un *string* con el nombre del *path* de la colección que queramos borrar. Después crearemos un *CollectionManagementService* que guardará la información que obtenga de la llamada que hará la colección a *getService*. Con este paso tendremos la información del servicio almacenada en la interfaz y podremos hacer la llamada al método *removeCollection*, pasándole el nombre de la colección que queramos eliminar. Si se produce algún error, el *catch* que tenemos en el código capturará el error con la clase *XMLDBException*. Tenemos que englobar nuestro código en un *try-catch* si queremos que el código compile correctamente y Eclipse no muestre ningún error en el código.

5.6. Añadir, modificar y eliminar documentos, clases y métodos

En este apartado, realizaremos una introducción de cómo añadir, modificar y eliminar documentos. A través de la API XML DB es posible desarrollar aplicaciones cliente que permiten mantener la estructura de las colecciones y recursos que establecen los sistemas nativos XML. En las secciones anteriores os hemos enseñado a crear y borrar una colección tanto en código Java como a través del programa de escritorio de eXist DB, ahora aprenderemos a añadir, modificar y eliminar los documentos de una colección.

5.6.1. Clases y métodos para añadir, modificar y eliminar documentos

Para la gestión de ficheros tenemos dos tipos de clases que se encargarán de realizar las gestiones:

XMLResource

Como hemos explicado en apartados anteriores, un documento será el fichero XML que queramos guardar en la base de datos. Para ello, utilizaremos la clase *XMLResource:* esta es una interfaz proporciona acceso al fichero XML que tenemos almacenado en la base de datos.

Los métodos más importantes de esta interfaz y con los que tenemos que estar familiarizados son:

Tipo	Método	Descripción
org.w3c.dom.Node	getContentAsDOM()	Este método devuelve el contenido del recurso en formato DOM Node.
void	getContentAsSAX(org.xml.sax.ContentHandler handler)	Permite usar un *ContentHandler*, analizar los datos XML de la base de datos para usarlos en una aplicación.

String	getDocumentId()	Este método se encarga de devolver el id único de un documento o *null* si no encuentra ningún tipo de información sobre el recurso padre.
boolean	getSAXFeature(String funcion)	Devuelve la configuración actual de una función SAX que se usará cuando este *XMLResource* se use para producir eventos SAX a través del método *getContentAsSAX()*.
void	setContentAsDOM(org.w3c.dom.Node contenido)	Este método setea el contenido del recurso utilizando un nodo DOM como fuente.
org.xml.sax.ContentHandler	setContentAsSAX()	Se encarga de setear el contenido del recurso utilizando un SAX *ContentHandler*.
void	setSAXFeature(String funcion, boolean valor)	Establece una función SAX que se usará cuando

		este *XMLResource* se use para producir eventos SAX a través del método *getContentAsSAX ().*

Resource

Esta es una interfaz que se utiliza como contenedor para guardar los datos que se obtienen del documento en la base de datos.

Tipo	Método	Descripción
Object	getContent()	Devuelve el contenido de un *Resource*.
String	getId()	Devuelve el identificador del recurso o *null* si no encuentra nada o si es anónimo.
Collection	getParentCollection()	Este método devuelve una instancia de la colección con la que el recurso está asociado.
String	getResourceType()	Devuelve el tipo del recurso que se esté consultando.
Void	setContent()	Setea el contenido de un recurso.

5.6.2. Añadir un documento

En la API XML DB, un documento es considerado un recurso, tal y como hemos explicado en apartados anteriores. En Java, para poder gestionar este tipo de documentos, utilizaremos *Resource* y *Collection*. Este último será necesario porque a través de la colección podremos indicar qué recursos podemos almacenar. Para entender mejor cómo se añade un recurso en la base de datos, vamos a desarrollar un ejemplo:

```
String URI = " xmldb:exist://localhost:8080/ilerna/";
Collection col =  DatabaseManager.getCollection(URI);
//Creamos un fichero con la ruta del XML que queremos guardar

File archivo = new File("libro.xml");
//Se obtiene el recurso
res = (Resource) col.createResource(archivo.getName(),
“XMLResource”);
res.setContent(archivo);
col.storeResource(res);
```

Tendremos que crear una instancia de una colección, que la usaremos para indicar en qué ruta se encuentra la colección que almacenará nuestros datos.

A continuación, tendremos que crear un archivo *File* al cual le asignaremos la ruta en la que se encuentra el fichero XML que queremos guardar.

Tras ello, con la colección creada, haremos una llamada a *createResource* pasándole por parámetro el nombre del fichero que guardar y el tipo de recurso. A continuación, setearemos el fichero al recurso. Finalmente, haremos una llamada al método de la colección *storeResource* pasándole el recurso que contiene la información del fichero XML que guardar.

5.6.3. Eliminar un recurso

Para eliminar un recurso, el proceso es bastante similar a los explicados en apartados anteriores. El método que se usa para eliminar un recurso es *removeResource*, y se llama a partir de una colección. La colección la utilizamos para indicar en qué ubicación de la base de datos se encuentra nuestro recurso. Si no indicamos la ruta correcta, no podremos borrar el fichero que nos interesa.

```
String URI = " xmldb:exist://localhost:8080/ilerna/";
Collection col =  DatabaseManager.getCollection(URI);

try{
      Resource recurso = null;
      File archivo = new File("libro.xml");

      //le asigna el contenid del archivo al nuevo recurso vacio
      recurso = collection.getResource(archivo.getName());
      coll.removeResource(recurso);
}catch(XMLDBException e){
      e.getMessage();
```

Para poder borrar un recurso, primero tenemos que crear una nueva instancia de *Collection* para poder guardar los datos de la colección donde se va a borrar el documento XML. A continuación, tenemos que crear un recurso que rellenaremos con el nombre del fichero que queremos buscar en la base de datos a partir de la colección, haciendo una llamada al método *getResource*. Eso recuperará la información del recurso de la base de datos que queremos borrar. Una vez recuperada la información, con la colección haremos un *removeResource* y le pasaremos el recurso. Todo el código irá envuelto en un *try-catch* para poder controlar los posibles errores que se puedan producir con *XMLDBException*.

5.6.4. Modificar un recurso

Para modificar un recurso, como hemos comentado en apartados anteriores, no existe manera de sustituir solo una parte del documento. Cuando queremos hacer una modificación de un recurso, tendremos que sustituir todo el documento entero. Por ello, lo primero que se debe hacer es buscar el recurso que se quiere sustituir, borrarlo y sustituirlo por el nuevo.

Para poder realizar esta acción, usaremos *Collection* para definir la ruta de la colección donde se encuentra el recurso y *Resource* para guardar la información del recurso que se encuentra en la base de datos.

```
String URI = " xmldb:exist://localhost:8080/ilerna/";
Collection col =  DatabaseManager.getCollection(URI);

Resource recurso = null;
File archivo = new File("libro.xml");

recurso = collection.getResource(archivo.getName());
col.removeResource(recurso);


File archivoGuardar = new File("coche.xml");
//Se obtiene el recurso
res = (Resource) col.createResource(archivo.getName(),
“XMLResource”);
res.setContent(archivoGuardar);
col.storeResource(res);
```

Como podemos ver en el ejemplo, la implementación es una mezcla de borrar un recurso. En este caso, buscamos el recurso que queremos sustituir y lo eliminamos. A continuación, tendremos que guardar el recurso nuevo, siguiendo los pasos que hemos comentado en el apartado 5.6.3.

5.7. Indexación, identificadores únicos

En este apartado, vamos a introducir el concepto de indexación en una base de datos nativa.

La **indexación** es una técnica para optimizar la obtención de registros de la base de datos.

Para mejorar el funcionamiento de consultas, las bases de datos XML nativas soportan la creación de índices para los datos almacenados en colecciones. Estos índices se pueden utilizar para mejorar la velocidad de ejecución de la consulta.

Cuando tenemos una gran cantidad de registros en la base de datos, realizar una consulta puede ser una tarea bastante costosa, ya que la base de datos tiene que ir recorriendo colecciones y sus documentos para poder encontrar aquella que se encuentra en la consulta. Por eso, la indexación desempeña un papel tan importante. Si se configura bien un índice, la consulta a la base de datos puede ser mucho más rápida. Estos archivos se guardan como documentos XML estándar en la colección del sistema:

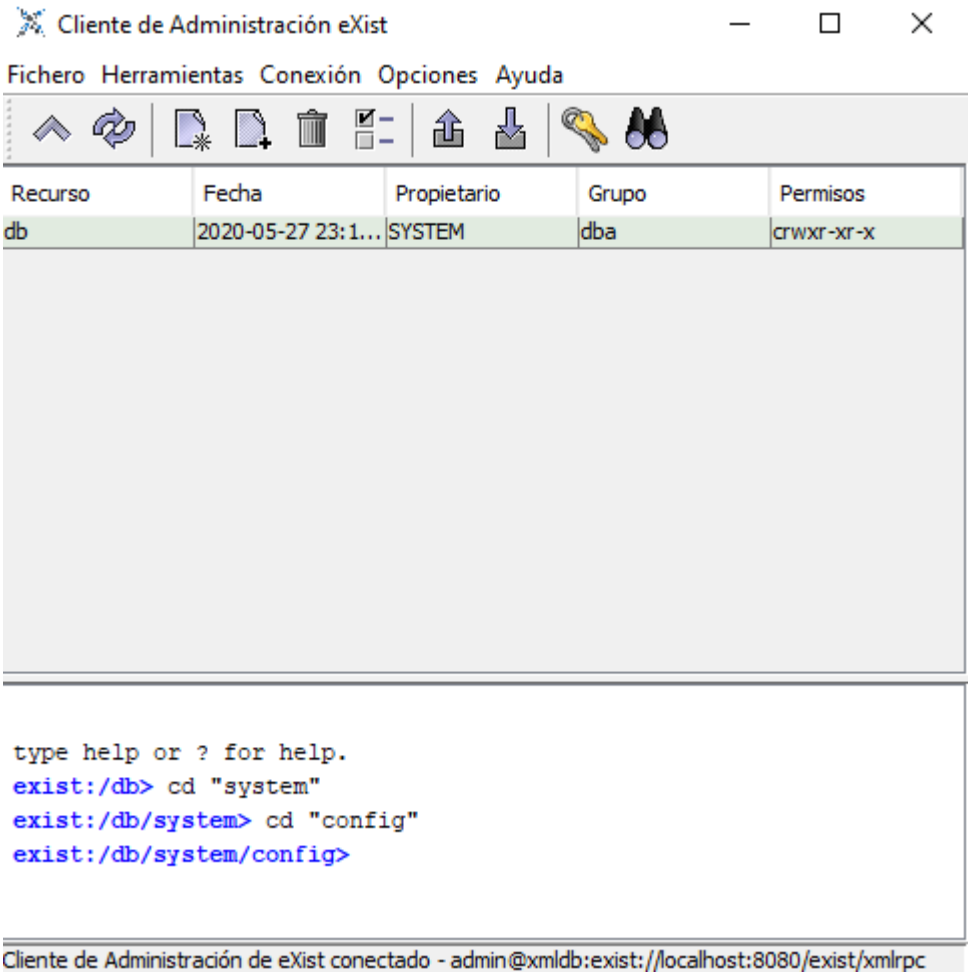
```
/db/system/config
```

En eXist, podemos diferenciar tres tipos de índices:

- **Índices estructurales**: se encargan de indexar la estructura de los nodos, los elementos y sus atributos en los documentos XML que se encuentran en una colección. Este tipo de índice se crea y se mantiene de manera automática por la base de datos.
- **Índices de texto completo o *fulltext*:** este tipo de índice se encarga de convertir los nodos y atributos en *tokens* de texto.
- **Índices de rango**: se basan en el tipo de dato de cada valor de los nodos del documento. Estos índices proporcionan un acceso directo para que la base de datos seleccione directamente nodos en función de estos valores de tipo. A diferencia de los índices estructurales y de texto completo, los índices pueden ser creados y configurados directamente por el usuario, y en este sentido son similares a los índices utilizados por las bases de datos relacionales.

Creación del fichero .xconf

En la base de datos eXist no existe un comando específico para crear un índice. Para poder realizar esta tarea, tenemos que dirigirnos a la configuración específica de las colecciones. Para acceder a la colección del sistema, lo podemos a través del cliente de Java o de la interfaz de administrador:



El contenido de la colección del sistema (/db/system/config) debe reflejar la estructura jerárquica de la colección principal. Las configuraciones son compartidas por los descendientes en la jerarquía, a menos que tengan su propia configuración: los ajustes de configuración para la colección secundaria anulan los establecidos para el padre. Si no se crea un archivo de configuración específico de la colección para ningún documento, la configuración global en el archivo de configuración principal conf.xml se aplicará de manera predeterminada. El archivo conf.xml solo debe definir la política de creación de índice global predeterminada.

Los ficheros de configuración de índices tienen la extensión .xconf y se almacenarán dentro de una colección con el mismo nombre del índice que usará esa configuración. Por ejemplo, si tenemos una colección llamada "ilerna" y dentro otra llamada "ejercicioIndice", el fichero de configuración se guardará de esta manera:

```
/db/system/config/ilerna/ejercicioIndice/collection.xconf
```

Si una colección o subcolección no dispone de este fichero de configuración, la búsqueda de los datos no se realizará con indexación, sino que se usará el método estándar de búsqueda de información.

Configuración del fichero .xconf

En esta sección, vamos a mostraros cómo se configura este tipo de fichero. Seguiremos la estructura de un índice de texto.

La estructura de un fichero .xconf está basada en un fichero XML y tiene sus elementos y atributos definidos por el espacio de nombres de eXist:

```
http://exist-db.org/collection-config/1.0
```

Vamos a ver en un ejemplo una muestra de un fichero ejemplo de configuración:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
    <index>
        <-Contenido de la configuración del indice ->
      <fulltext>

      </fulltext>

    </index>
</collection>
```

Todos los documentos de configuración deben empezar por un elemento *<collection>*. Directamente debajo del elemento raíz, colocaremos un elemento *<index>* que encierra la configuración del índice. Solo se permite un elemento *<index>* en cada fichero de configuración.

En el elemento *<index>* hay elementos que definen los diversos tipos de índice. Cada tipo de índice agrega sus propios elementos de configuración, que se envían directamente a la implementación del índice correspondiente.

A continuación, le sigue la etiqueta *<fulltext>* junto con los elementos *<include>* y *<exclude>*, que proporcionan rutas de índice específicas.

Los índices de rango están definidos por elementos *<create>*. Debemos tener en cuenta que un tag *<index>* contiene solo un elemento *<fulltext>*, pero puede contener más de un elemento *<create>* para ir estableciendo diferentes índices de rango.

Partiendo de este XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:x="http://www.foo.com">
    <item n="1"">
        <nombre>Red Bicycle</nombre>
        <precio precioespecial="false">645</precio>
        <stock>15</stock>
        <x:puntuacion>8.7</x:puntuacion>
    </item>
</items>
```

La configuración del índice sería algo parecido a esto:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
        <index xmlns:x="http://www.foo.com">
                <fulltext default="none" attributes="false"
alphanum="false">
                        <include path="//item/nombre"/>
                </fulltext>
                <create path="//item/@n" type="xs:integer"/>
                <create path="//item/nombre" type="xs:string"/>
                <create path="//item/stock" type="xs:integer"/>
                <create path="//item/precio" type="xs:double"/>
                <create path="//item/precios/@precioespecial"
type="xs:boolean"/>
                   <create path="//item/x:puntuacion"
type="xs:double"/>
        </index>
</collection>
```

Antes de explicar con detalle el ejemplo, tendremos que revisar la tabla con todas las etiquetas que se pueden utilizar en uno de estos ficheros de configuración de índices:

Etiqueta	Descripción
<collection>	Etiqueta raíz que determina que ese fichero es de configuración de índices.
<index>	Etiqueta que contiene la configuración de los índices.
<fulltext>	Esta etiqueta determina la configuración del tipo de indexación *fulltext* o de texto.

<include>	Etiqueta que especifica el nodo que se indexará de un documento en la indexación de tipo texto.
<exclude>	Etiqueta que especifica el nodo que se excluirá de la indexación de un documento en la indexación de tipo texto.
<create>	Esta etiqueta se encarga de definir el índice para un rango de nodos definidos en esa selección.

Con este ejemplo, el atributo por defecto de *<fulltext>* se establece en *"none"*, lo que deshabilita la indexación de texto completo predeterminado para todos los elementos del documento. La excepción a esto son los elementos de nombre que usan el elemento *<include>*.

Podemos observar también que cada elemento *<create>* tiene un atributo de ruta que define los nodos a los que se aplica la configuración y que se expresan como rutas de índice. A continuación, cada *<include>* tiene un atributo *path* al que se le indicará mediante XPath el elemento que se debe indexar. Los índices de rango son específicos para cada tipo de dato que corresponda a un nodo: si definimos un tipo de dato que no corresponde al nodo que hemos definido, ese índice se ignorará.

5.8. Realización de consultas, clases y métodos

En este apartado, haremos una introducción al principal lenguaje de consultas para las bases de datos XML: **XPath** (XML Path Language). Tal y como comentábamos en el apartado 5.1, XPath es un lenguaje de consultas que se utiliza para consultar datos de un fichero XML, centrado en bases de datos XML nativas. Este lenguaje se caracteriza por:

- XPath utiliza la sintaxis de rutas para identificar y navegar por los nodos de un documento XML.
- Este lenguaje contiene más de doscientas funciones integradas.
- XPath es un elemento importante en el estándar XSLT. XSLT significa XSL Transformations, que se basa en XSL (eXtensible Stylesheet Language), un lenguaje para dar formato a los documentos XML.
- Este lenguaje se definió por parte del consorcio W3C. Este consorcio se encarga de realizar recomendaciones y estándares para asegurar el crecimiento y que se sigan unas normas de la World Wide Web.

Por otro lado, un aspecto importante de este lenguaje es que está diseñado para ser un lenguaje de consulta y no está pensado para ser utilizado como lenguaje de modificación de datos, lo que lo hace un lenguaje limitado. Las limitaciones más destacables que debemos tener en cuenta son:

- Falta de sentencias para realizar agrupaciones *(group by)*, ordenar *(order by)* o enlazar diferentes documentos *(join)*.
- No soporta la gestión de datos de tipos distintos.

A causa de las limitaciones, este lenguaje se tiene que complementar para poder ampliar sus capacidades. Como hemos explicado en las características de este lenguaje, XPath puede ser complementado con las plantillas XSL y XSLT.

Haremos una breve introducción a los elementos básicos para realizar consultas, sus sentencias, cómo crear condiciones y las principales clases y métodos para gestionar este tipo de consultas en Java.

5.8.1. Lenguajes de consulta suministrados por el gestor de bases de datos XPath

XPath se basa en la utilización de expresiones definidas a través de un patrón para poder seleccionar un conjunto de nodos de un documento XML. XSLT utiliza esos patrones para transformar el documento.

La especificación XPath detalla siete tipos de nodos que pueden ser el resultado de la ejecución de la expresión XPath. Estos son algunos ejemplos de la terminología que se utiliza en XPath:

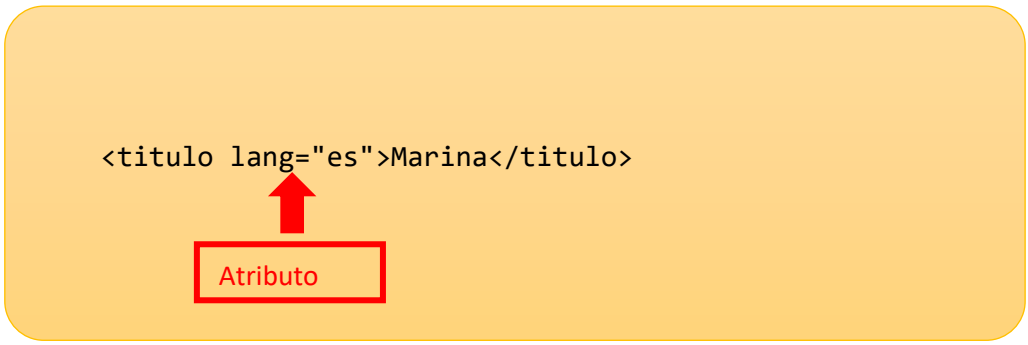
- Raíz.
- Elemento.

```
<?xml version="1.0" encoding="UTF-8"?>

<biblioteca>
  <libro>
    <titulo lang="es">Marina</titulo>
    <autor>Carlos Ruiz Zafon</autor>
    <precio>29.99</precio>
  </libro>
</biblioteca>
```


- Texto.

- Atributo.



- Valores atómicos: nodos que no tienen hijos o padre. En este caso, podría ser el nombre del lenguaje: "es".
- Espacio de nombres.

Relación entre nodos

Para entender mejor cómo funciona este lenguaje, vamos a explicar cómo se relacionan los diferentes elementos del XML y qué nombre utilizaremos para referirnos a ellos.

- Elemento padre: se refiere a la etiqueta que define el elemento que queremos representar. Por ejemplo, en el XML del ejemplo anterior un elemento padre sería libro.
- Elemento hijo: son todos los nodos que se encuentran dentro de un elemento. Siguiendo con el ejemplo anterior, los elementos hijos serían "titulo", "autor" o "precio". Estos elementos hijo serán hermanos unos de otros (lo conocido como *siblings*). Un nodo puede no tener hijos o tener más de uno.
- Elemento antepasado *(ancestor):* este es el elemento que se encuentra por encima del elemento padre. Siguiendo con el ejemplo, se podría considerar elemento antepasado "biblioteca".
- Elemento descendiente *(descendants):* serían todos los elementos que se encuentren dentro de la etiqueta *ancestor*. En este caso, todos los elementos "libro" y todos los elementos que se encuentren dentro de "libro".

Selección de nodos

Para realizar la búsqueda de información dentro del XML, XPath utiliza expresiones de ruta para seleccionar un nodo o una lista de nodos.

A continuación, mostraremos una lista de expresiones que nos serán útiles para aprender el lenguaje XPath, y sobre todo para seleccionar cualquier lista o nodo de un documento:

Expresión	Descripción
/	La selección empezará desde la raíz del nodo. Ejemplo: */biblioteca.*
//	La selección comienza desde el nodo actual que coincide con la selección. Ejemplo: ***//libro*** selecciona todos los elementos que se llamen "libro".
.	Expresión que se encarga de seleccionar el nodo actual.
..	Expresión que se encarga de seleccionar el nodo padre del nodo actual.
@	Expresión que selecciona los atributos. Ejemplo: ***//@lang*** selecciona todos los atributos que se llamen "lang".
nombre_nodo	Selecciona todos los nodos con el nombre que indiquemos. Ejemplo: **"libro"**.
nodoPadre/nodoHijo	Selecciona todos los nodos hijo que se encuentren dentro del nodo padre. Ejemplo: ***biblioteca/libro***
//nombre_nodo	Seleccionará todos los elementos que indiquemos que se encuentren en el documento.

Predicados

Los predicados son otra parte importante de este lenguaje. Se utilizan para encontrar nodos de manera más específica o también el valor que se encuentra dentro de un nodo. Los predicados siempre se incluyen entre corchetes. Vamos a mostraros en una tabla las expresiones más importantes y más útiles que podemos utilizar como predicados:

Expresión	Descripción
/biblioteca/libro[1]	Este predicado se encargará de seleccionar el primer elemento "libro" que sea un elemento hijo de "biblioteca".
/biblioteca/libro[last()]	Este predicado se caracteriza por seleccionar el último elemento "libro" que sea hijo del elemento "biblioteca". Para seleccionar el último elemento, utilizamos *last()*.

/biblioteca/libro[last()-1]	Esta expresión se encarga de devolver el penúltimo elemento "libro" que sea hijo del nodo "biblioteca".
/biblioteca/libro[position()<3]	Selecciona los primeros dos elementos "libro" que sean hijos de "biblioteca".
//titulo[@lang]	Seleccionará el título de los elementos que tengan como atributo "lang".
//titulo[@lang='es']	Seleccionará el título de los elementos que tengan un atributo "lang" con valor "es".
/biblioteca/libro[precio>35]	Seleccionará todos los elementos "libro" que estén dentro de "biblioteca" y que tengan un precio con un valor mayor a 35.
/biblioteca/libro[precio>35]/titulo	Selecciona todos los elementos "titulo" dentro del elemento "libro" que se encuentren dentro del elemento "biblioteca" y que tengan una etiqueta "precio" cuyo valor sea mayor de 35.

Los predicados nos servirán para filtrar de manera precisa los datos que queremos seleccionar. La tabla solo muestra los ejemplos más sencillos, pero como vemos en la última expresión lo podemos complicar para que seleccione de manera más precisa los datos. Utilizar este tipo de predicado es especialmente útil cuando un XML tiene más de un elemento "libro", por ejemplo.

Nodos desconocidos

Seguramente habrá ocasiones en que no sabremos qué elementos queremos seleccionar. Para ello, vamos a mostrar ejemplos de cómo realizar búsquedas cuando los nodos que queremos seleccionar son desconocidos.

Expresión	Descripción
*	Selecciona cualquier elemento nodo. Ejemplo: ***/biblioteca/**** seleccionará todos los hijos de "biblioteca".
@*	Selecciona cualquier atributo.
node()	Selecciona cualquier nodo de cualquier tipo.

Selección de múltiples *path*

XPath también permite el operador **|** que nos va a permitir crear más de una expresión y ejecutarla en la misma consulta, solamente tenemos que crear dos consultas y unirlas con ese operador. Vamos a mostrar unos ejemplos:

//libro/titulo | // libro/precio: seleccionará todos los títulos y precios de los elementos "libro".

//titulo | //precio: seleccionará todos los elementos título y los elementos precio del documento.

Ejes

Un eje representa una relación con el nodo de contexto (actual) y se usa para ubicar nodos relativos a ese nodo en la estructura del árbol. A la hora de realizar consultas, estas relaciones nos serán útiles para ir seleccionado los diferentes elementos del XML. Vamos a ver una tabla esquema con los ejes más importantes y sus tareas principales:

Eje	Descripción
ancestor	Selecciona todos los ancestros (padre, abuelo…) del nodo actual.
ancestor-or-self	Selecciona todos los ancestros (padre, abuelo…) del nodo actual y el del mismo nodo.
attribute	Selecciona todos los atributos del nodo actual.
child	Selecciona todos los hijos del nodo actual.
descendant	Selecciona todos los descendientes del nodo actual.
descendant-or-self	Selecciona todos los descendientes del nodo actual y el propio nodo.
following	Selecciona todo lo que encuentra en el documento después del *tag* que cierra el nodo actual.
following-sibling	Selecciona todo lo que encuentra en el documento después del *tag* que cierra el nodo actual, así como también el propio nodo.
namespace	Selecciona todos los *namespaces* del propio nodo.
parent	Selecciona todos los nodos padre del propio nodo.

preceding	Selecciona todos los nodos que aparecen antes del nodo actual en el documento excepto antepasados, nodos de atributos y nodos de espacio de nombres.
preceding-sibling	Selecciona todos los hermanos antes del nodo actual.
self	Selecciona el nodo actual.

Rutas

Como explicamos en el tema 1, una ruta puede ser absoluta o relativa.

Una ruta absoluta empieza con una barra (/), y una ruta relativa irá sin barra. En ambos casos, la ruta de localización del fichero se irá formando con el nombre del elemento y una barra:

Ruta relativa: biblioteca/libro/

Ruta absoluta: /biblioteca/libro/

Operadores

XPath también permite realizar consultas a partir de operadores, permite los más usuales, como en SQL. Vamos a mostrar un resumen de todos ellos:

Operador	Descripción
|	Calcula dos conjuntos de nodos.
+	Suma.
-	Resta.
*****	Multiplicación.
Div	División.
=	Igual.
!=	No igual.
<	Menos que.
>	Más que.

<=	Menos o igual que.
>=	Más o igual que.
or	O.
and	Y.
mod	Módulo.

Implementación en una aplicación Java

Para realizar una consulta con XPath a través de Java, tendremos que introducir nuevas clases que nos ayudarán a ejecutar esta tarea:

- *XPathFactory:* clase que sirve para crear objetos XPath.
- *XPath:* es la clase que necesitaremos para poder crear posteriormente nuestra expresión. Nos será útil de auxiliar para crear el *XPathExpresion*.
- *XPathExpresion:* la utilizaremos para guardar la información de la sentencia XPath que queremos que se ejecute en la base de datos.

```
//Creamos la estructura para trabajar con el XML
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse("libro.xml");

//Creamos el XPath

XPathFactory xpathfactory = XPathFactory.newInstance();
XPath xpath = xpathfactory.newXPath();

System.out.println("n//1) Recogemos los titulos de los libros con
precio mayor de 20 euros");

      XPathExpression expr =
xpath.compile("//libro[@precio>20]/titulo/text()");
      Object result = expr.evaluate(doc, XPathConstants.NODESET);
      NodeList nodes = (NodeList) result;
      for (int i = 0; i < nodes.getLength(); i++) {
          System.out.println(nodes.item(i).getNodeValue());
      }
```

En primer lugar, tenemos que crear un documento para poder guardar el recurso que queremos utilizar, en esta ocasión es libro.xml, que contendrá esta información dentro:

```
<?xml version="1.0" encoding="UTF-8"?>

<biblioteca>
  <libro>
    <titulo lang="es">Marina</titulo>
    <autor>Carlos Ruiz Zafon</autor>
    <precio>29.99</precio>
  </libro>
</biblioteca>
```

Este fichero libro.xml lo tendremos en nuestro ordenador o en un servidor, y cuando tengamos que crear el documento, le pasaremos la ruta donde se guarda el fichero. Para crear una nueva instancia de XPath, primero tendremos que crear un *XPathFactory* y una nueva instancia con *newInstance*. A continuación, creamos una instancia de Xpath con *xpathFactory.newXPath()*. Esta instancia será la que usemos para hacer las búsquedas dentro del documento que hemos cargado. Seguidamente, tendremos que crear un *XPathExpresion*, que nos servirá para crear la sentencia con el lenguaje XPath. El método *compile* servirá para cargar la información de la expresión a la nueva instancia creada. Con *expr.evaluate(doc, XPathConstants.NODESET)* ejecutaremos la consulta y se guardarán los datos a un *Object*. Este guardará el listado de resultados que, para poder ser tratado, necesitaremos hacer un *cast* a un *NodeList*. Una vez que tengamos el *NodeList*, podremos tratar los datos de la manera que deseemos.

5.8.2. Gestión de transacciones

Para la gestión de transacciones en bases de datos XML, se introduce la interfaz *TransactionService*. Para hacer memoria, una transacción es un conjunto de órdenes que permiten ejecutar una consulta en la base de datos.

Los métodos que más utilizaremos de esta interfaz son:

Tipo	Método	Descripción
void	begin()	Método que se encarga de inicializar una transacción.
void	commit()	Método que ejecuta la transacción.

void	rollback()	Se encarga de tirar atrás todas las acciones que se han realizado hasta el momento.
void	setTimeout(int segundos)	Este método establece un *timeout* a la transacción: si durante el tiempo establecido no se ha podido ejecutar la acción, se lanza un *timeout*.
boolean	isAlive()	Verifica si la transacción aún está activa.
boolean	wasCommited()	Verifica si la transacción se ha comiteado correctamente.
boolean	wasRolledBack()	Verifica si la transacción ha realizado *rollback* correctamente.

Para poder ver mejor cómo implementar esta interfaz y que funcione correctamente, vamos a ver un ejercicio con la explicación:

```
Document document1;
Document document2;

String id1 = "gladiator-2000";
String id2 = "gonein60seconds-2000";

TransactionService transaction =
(TransactionService) collection.getService("TransactionService",
"1.0");

transaction.begin();

XMLResource resource1 =
(XMLResource)collection.createResource(id1,
XMLResource.RESOURCE_TYPE);

resource1.setContentAsDOM(document1);
collection.storeResource(resource1);

XMLResource resource2 =
(XMLResource) collection.createResource(id2,
XMLResource.RESOURCE_TYPE);

resource2.setContentAsDOM(document2);
collection.storeResource(resource2);

transaction.commit();
```

Para gestionar las transacciones en la base de datos eXist, haremos uso del servicio *TransactionService*. Este servicio nos permite obtener la información de la base de datos de las transacciones y tener control en el código. El proceso de creación es parecido al expuesto en apartados anteriores. A través de la colección, haremos una llamada al servicio *TransactionService* y guardaremos la información a una nueva instancia de *TransactionService*. Una vez que tengamos la clase con la información de la base de datos, ya podremos trabajar con las transacciones. Lo primero antes de realizar ninguna ejecución sobre la base de datos es hacer un *transaction.begin()*, este método nos va a permitir controlar las transacciones a lo largo del código de manera manual. Para que se ejecuten todas las consultas o acciones contra la base de datos, tendremos que utilizar el método *commit()*. Si no se produce un error antes, al realizar *commit()* se ejecutarán todas las sentencias o acciones.

http://xmldb-org.sourceforge.net/xapi/UseCases.html

5.9. Lenguaje de consulta para XML: XQuery (XML Query Language)

En este apartado realizaremos una introducción al lenguaje de consultas para bases de datos XML. Existen más opciones de lenguaje de consulta, pero nos centraremos en XQuery.

El lenguaje **XQuery** (XML Query Language) es un tipo de lenguaje de consulta propuesto por la W3C. Está enfocado para bases de datos XML y ofrece la posibilidad de la búsqueda de datos para colecciones.

Se trata de un lenguaje muy similar a SQL e incluye algunas capacidades de programación. Este lenguaje surge como un equivalente al lenguaje SQL y está destinado al uso en bases de datos XML. Está considerado como una extensión de XPath 2.0.

Las características principales de este lenguaje son:

- **Funcionalidad**: es un tipo de lenguaje muy funcional que está enfocado en realizar consultas en bases de datos XML.
- **Parecido a SQL**: es un lenguaje que puede definirse como el equivalente a SQL, pero para bases de datos XML.
- **Basado en XPath**: es un lenguaje que está basado en otro, XPath, que es el principal lenguaje para navegar a través de documentos utilizando expresiones.
- **Universalidad**: es un lenguaje que se acepta universalmente para todo tipo de bases de datos.
- **Estándar W3C**: este lenguaje sigue el estándar W3C.

Los beneficios que nos ofrece este tipo de lenguaje son:

- Recuperación de datos jerárquicos y tabulares.
- Realización de consultas con estructuras de árbol y gráficas.
- XQuery se puede usar directamente para consultar y crear páginas web.
- Es capaz de transformar documentos XML.
- XQuery es ideal para bases de datos basadas en XML y bases de datos basadas en objetos. Las bases de datos de objetos son mucho más flexibles y potentes que las bases de datos puramente tabulares.

Para poder realizar una consulta XQuery, primero tenemos que aprender cómo funciona este lenguaje. Este lenguaje está basado en XPath, por tanto, todas las expresiones, operadores y características en este lenguaje serán iguales.

Funciones

Xquery utiliza funciones para realizar las consultas de los documentos XML. Una sentencia básica de búsqueda en un fichero XML sería así:

```
doc(“nombreFichero.xml”)/biblioteca
```

Como vemos, se empieza definiendo en qué fichero queremos realizar la búsqueda utilizando *doc()* y pasándole por parámetro el nombre del fichero. A continuación, ya sería parecido a XPath para refinar la búsqueda. Este lenguaje utiliza las expresiones de ruta para poder navegar entre los elementos del documento, tal y como vimos con XPath.

También utiliza los predicados para poder refinar más las búsquedas. En este ejemplo podremos ver cómo utilizamos un predicado para indicar que queremos buscar todos los libros que se encuentran dentro de "biblioteca" y que tengan un precio mayor de 35 euros.

```
doc(“nombreFichero.xml”)/biblioteca/libro [precio>35]
```

XQuery FLWOR

Esta es una estructura ya más elaborada. FLWOR hace referencia a *for*, *let*, *where*, *order by* y *return*.

- **For**: para seleccionar la secuencia de nodos.
- **Let**: para unir una secuencia a una variable.
- **Where**: para filtrar los nodos.
- **Order by**: para ordenar los nodos.
- **Return**: para especificar qué se tiene que devolver como resultado.

La estructura FLWOR se crea de esta manera:

```
for $x in doc(“libros.xml”)/biblioteca/libro
where $x/precio>35
order by $x/titulo
return $x/titulo
```

Esta sentencia utiliza *for* para seleccionar la información de los elementos, y definimos una variable *$x*. Como vemos, aquí se introducen las variables y se definen con el símbolo del dólar y su nombre, en este caso, una *x*. A continuación, añadimos la cláusula *where*, que nos servirá para refinar la búsqueda solo a los libros que tengan como precio un número mayor de 35. Seguidamente, indicamos con el *order by* que queremos ordenar los valores por título. Finalmente, indicamos que el valor que retornar será la variable *x* con los títulos.

Como se puede comprobar, es un lenguaje bastante fácil de aprender, porque mezcla conocimientos de SQL con el XPath que hemos visto en el apartado anterior.

Reglas de las sintaxis

Para poder realizar correctamente las sentencias, debemos regirnos por unas normas:

- Es sensible a mayúsculas.
- Se deben utilizar nombres válidos para los elementos, atributos y variables.
- Se pueden utilizar comillas simples o dobles, pero no mezclarlas.
- Las variables se definen con el símbolo del dólar ($) seguido del nombre que se quiera utilizar.
- Se pueden utilizar comentarios, tienen que empezar con dos puntos (:) y terminar con dos puntos.

Expresiones condicionales

Para poder crear un condicional en XQuery, se utiliza la expresión *if-then-else*. Para establecer la condición se utilizan paréntesis, como en muchos lenguajes de programación. Para entender mejor cómo realizar esta expresión en una consulta, vamos a ver un ejemplo de su implementación:

```
for $x in doc("libro.xml")/biblioteca/libro
return if ( $x/@lang="es" )
then <castellano>{data($x/titulo)}</castellano>
else <ingles>{data($x/titulo)}</ingles>
```

La estructura que se sigue es *for* sin el resto de cláusulas más que el *return*. En el *return* se define el condicional. Si se cumple la cláusula, se ejecutará el *then;* de lo contrario, pasará al *else*.

Creación de secuencias

En XQuery las sentencias se usan con los paréntesis () y dentro contendrán *strings* con los parámetros. Estos *strings* se pueden pasar con comillas simples o dobles, pero no se pueden mezclar las dos.

En esta tabla vamos a mostrar las secuencias más importantes que podemos utilizar:

Función	Descripción
count($seq as ítem()*)	Cuenta los ítems de una secuencia.
sum($seq as ítem()*)	Devuelve la suma de los ítems de una secuencia.
avg($seq as ítem()*)	Devuelve la media de los ítems de una secuencia.
min($seq as ítem()*)	Devuelve el mínimo valor de los ítems de una secuencia.
max($seq as ítem()*)	Devuelve el máximo valor de los ítems de una secuencia.
distinct-values($seq as ítem()*)	Devuelve una selección de ítems no iguales entre ellos de una secuencia.
subsequence($seq as ítem()*, $startingLoc as xs:double, $length as xs:double)	Devuelve un subconjunto de la secuencia que indiquemos por parámetro.
Insert-before($seq as ítem()*, $position as xs:integer, $inserts as item()*)	Introduce un ítem dentro de una secuencia.
remove($seq as ítem()*)	Elimina un ítem de una secuencia.
index-of($seq as anyAtomicType()*, $target as anyAtomicType())	Devuelve índices como enteros para indicar la disponibilidad de un artículo dentro de una secuencia.

last()	Devuelve el último elemento de una secuencia cuando se usa en la expresión predicada.
position	Se usa en las expresiones FLOWR para conseguir la posición de un ítem en una secuencia.

Funciones de tipo secuencia más específicas de los *strings*

Función	Descripción
string-length($string as xs:string) as xs:integer	Devuelve el tamaño del *string*.
concat($input as xs:anyAtomicType?) as xs:string	Se encarga de concatenar *strings*.
string-join($sequence as xs:string*, $delimiter as xs:string) as xs:string	Combina ítems dentro de la misma secuencia separados por un identificador.

Expresiones regulares

XQuery se caracteriza también por el uso de expresiones regulares, esto puede ser especialmente útil para poder encontrar elementos que coincidan con una expresión que le indiquemos a la función. Vamos a ver las funciones de expresiones regulares más importantes:

Función	Descripción
matches($input,$regex)	Devuelve un *true* si el *input* coincide con la expresión regular
Replace($input,$regex, $string)	Se encarga de sustituir un *input* que coincida con un *string* que le pasamos por parámetro.
tokenize($input,$regex)	Devuelve una secuencia de ítems que coinciden con la expresión regular.

Ejecución de sentencias XQuery en Java

Para poder ejecutar XQuery en Java primero tendremos que preparar nuestro proyecto. Para la ejecución de sentencias con este lenguaje, tendremos que utilizar la API de XQJ, esta librería está especialmente pensada para la gestión de bases de datos XML con un lenguaje XQuery.

Para que nuestro proyecto funcione correctamente, tendremos que añadir estas dos librerías:

https://maven.repository.redhat.com/ga/net/sf/saxon/saxon9he/9.4.0.4/saxon9he-9.4.0.4.jar

https://repo.fusesource.com/nexus/content/repositories/releases-3rd-party/net/sf/saxon/saxon-xqj/9.1.0.8/saxon-xqj-9.1.0.8.jar

Para la ejecución de este ejemplo tendremos que tener preparado:

- Un fichero .xml llamado libros.xml que contendrá el XML con los datos de los libros que hemos usado a lo largo de los ejemplos de este tema. El contenido del XML será este:

```
<?xml version="1.0" encoding="UTF-8"?>

<biblioteca>
  <libro>
    <titulo lang="es">Marina</titulo>
    <autor>Carlos Ruiz Zafon</autor>
    <precio>29.99</precio>
  </libro>
</biblioteca>
```

- Un fichero que llamaremos libro.xqy, que contendrá la consulta en lenguaje XQuery. En el contenido de este fichero, realizaremos una consulta simple en XQuery, como esta:

```
<result>
{
   if(not(doc("libros.xml"))) then (
      <error>
         <message>El fichero libros.xml no
existe</message>
      </error>
   )
   else (
      for $x in doc("libros.xml")/books/book
      where $x/precio>30
      return $x/titulo
   )
}
```

Para realizar un fichero de consulta .xqy, tendremos que crear un XML con la raíz *<result>*, que se utiliza para indicar que el contenido de esa ejecución será el resultado de la consulta XQuery. Dentro, podremos utilizar las expresiones que hemos ido explicando a lo largo de este apartado. Realizamos una comprobación de si existe el fichero libros.xml: si no existe, lanzará un mensaje de error. El error se encuentra en un elemento *<error>*, y el mensaje en un elemento hijo *<message>*. Si encuentra el fichero, buscará todos los elementos libro con un valor de más de 30 y retornará el título.

- Una clase Java que hará la consulta a la base de datos.
  https://www.progress.com/tutorials/xquery/api-for-java-xqj

```
try {
      //Definimos el fichero de consulta
      InputStream consultaXQuery = new FileInputStream(new
File("libro.xqy"));
      //Definimos la conexión
      XQDataSource ds = new SaxonXQDataSource();
      Connection conexion =
DriverManager.getConnection("xmldb:exist://localhost:8080/ilerna/"
, "admin", "root");
      XQConnection conn = ds.getConnection(conexion);

      XQPreparedExpression exp =
conn.prepareExpression(consultaXQuery);
      XQResultSequence result = exp.executeQuery();

      while (result.next()) {
      System.out.println(result.getItemAsString(null));
      }
} catch (FileNotFoundException e) {
      e.printStackTrace();
} catch (XQException e) {
      e.printStackTrace();
} catch (SQLException e) {
      e.printStackTrace();
}
```

Como podemos ver en el ejemplo, las clases para gestionar las conexiones y para realizar las consultas son algo distintas de las que hemos visto hasta ahora.

- ***XQDataSource**:* es una interfaz que nos va a permitir a preparar los objetos *XQConnection* para preparar la conexión con la base de datos.
- ***XQConnection**:* es la interfaz que establece la conexión con la base de datos, y almacena en esta clase toda la información referente a esta conexión. Para ello, necesita que al método *getConnection* le pasemos una instancia de *Connection* con los datos de la URL de la base de datos, el usuario y la contraseña.
- ***XQPreparedExpression**:* esta clase nos va a ayudar a preparar el contexto para poder realizar la consulta con XQuery. Si nos fijamos en el ejemplo, con la conexión hace una llamada a *prepareExpression()*, recibiendo por parámetro el fichero que hemos creado al principio con la *query* en el lenguaje Xquery. El objeto preparará todo el contexto para que con el *XQResultSequence* podamos ejecutar la consulta y almacenar los resultados.
- ***XQResultSequence**:* es la clase que recoge los resultados de la consulta con XQuery. Esta clase nos va a permitir gestionar todos los resultados.
- ***XQException**:* esta clase de error gestiona los posibles errores que se producen durante el proceso de ejecución de una consulta a la base de datos.

5.10. Tratamiento de excepciones

Con la API XML DB, las excepciones se recogen con ***XMLDBException***. Esta interfaz se encarga de capturar todos los errores que se produzcan cuando se ejecuta el código bajo esta API. Contiene dos tipos de errores: el que se define dentro de *ErrorCodes* y otro que se especifica en el sistema de la base de datos XML nativa, en este caso, eXistDB. Si el error que se genera es específico del proveedor de la base de datos, entonces el código de error tiene que establecerse en *ErrorCodes.VENDOR_ERROR*.

La característica de esta interfaz de gestión de errores es que, en lugar de utilizar una clase para cada tipo de error, lo que hace es lanzar un código de error específico para cada tipo de error que se pueda producir. Algunos de los errores de esta clase son:

- *COLLECTION_CLOSED:* se activa para indicar que la colección está cerrada.
- *INVALID_DATABASE*, *INVALID_COLLECTION*, *INVALID_RESOURCE*, *INVALID_URI:* se activan para indicar que la base de datos, la colección, el recurso o un URI son inválidos.
- *NO_SUCH_DATABASE*, *NO_SUCH_COLLECTION*, *NO_SUCH_RESOURCE* y *NO_SUCH_SERVICE:* se lanzarán cuando una colección, la base de datos, un recurso o un servicio no se pueda encontrar.
- Otros errores: *NOT_IMPLEMENTED*, *PERMISSION_DENIED*, *UNKNOWN_ERROR*, *UNKNOWN_RESOURCE_TYPE*, *WRONG_CONTENT_TYPE*.

En el siguiente recuadro tenemos un ejemplo de implementación del *XMLDBException*.

```
try {
    XMLResource r = (XMLResource) this.col.createResource(id,
          "XMLResource");
    this.col.removeResource(r);
    this.col.close();
} catch (XMLDBException e) {
    if(e.errorCode == ErrorCodes.INVALID_RESOURCE) {
       /* Ok. If for any reason the message was not found there
is no
        * need to delete it.
        * */
    } else {
       throw new WNSException("XMLDBException" +
e.getLocalizedMessage());
    }
}
```

Para poder lanzar un error, el código debe estar envuelto en un *try-catch*. Si el código que tenemos no puede borrar un recurso, se producirá el error, que se capturará en el *catch*. Dentro del *try*, podemos ver que usamos la clase *ErrorCodes* para comprobar si el código de error es un recurso inválido. Podremos personalizar el *catch* con mensajes según el tipo de error que se produzca.

6. Programación de componentes de acceso a datos

En este tema profundizaremos en la programación orientada a componentes. Actualmente, los avances que se están produciendo en la informática están haciendo evolucionar la forma en que se desarrollan las aplicaciones de *software*. En concreto, el gran aumento de la potencia de los ordenadores, el abaratamiento de los costes del *hardware* y las comunicaciones y también la aparición de grandes redes de datos de cobertura global han hecho aumentar el uso de los sistemas abiertos y distribuidos. Esto ha desencadenado que los modelos de programación existentes se vean desbordados a causa de la gran cantidad de requisitos que exigen este tipo de sistemas. Debido a todo esto, se planteas nuevas metodologías de programación, como la programación orientada a componentes, con el objetivo de mejorar el proceso de construcción de las aplicaciones, pero también de desarrollar código que sea reutilizable.

Podemos entender como programación orientada a componentes lo siguiente:

> La **programación orientada a componentes** es una rama de la ingeniería de *software* que se enfoca en la programación de módulos de *software* reutilizables.

El objetivo de este tipo de programación era poder solucionar las exigencias de estos nuevos modelos de desarrollo y facilitar el trabajo de desarrollo de aplicaciones para poder realizarlas en menor tiempo. Es por ese motivo que se empezaron a construir aplicaciones a partir de componentes ya existentes, ya fueran componentes comerciales o componentes de *software* de código libre.

Uno de los enfoques en los que actualmente se trabaja constituye lo que se conoce como **desarrollo de *software* basado en componentes (DSBC)**, que trata de sentar las bases para el diseño y desarrollo de aplicaciones distribuidas basadas en componentes *software* reutilizables. Dicha disciplina cuenta actualmente con un creciente interés, tanto desde el punto de vista académico como desde el industrial, donde la demanda en este aspecto es cada día mayor.

Podemos considerar este tipo de programación una variante de la programación orientada a objetos.

6.1. Concepto de componente; características

Para entender mejor la programación orientada a componentes, debemos tener claro qué es un componente y cuáles son sus características principales.

Un **componente** se podría definir como una parte o módulo de una aplicación que está formado por un conjunto de dependencias e interfaces y que añade funcionalidad a un *software*.

En resumen, un componente es un módulo que se ejecuta de manera independiente como complemento de una aplicación, con una funcionalidad e interacción perfectamente definidas. Se puede añadir sin tener que modificar ningún otro componente de la aplicación. Además, un componente es perfectamente sustituible por otro de una funcionalidad igual o similar en cualquier momento.

Un claro ejemplo de un componente podría ser un *plugin* o una extensión para el navegador. Es una funcionalidad que proporciona características adicionales, no es necesaria su modificación para que funcione, es independiente, interactúa con el navegador y en cualquier momento podemos sustituirla por otra que efectúe la misma tarea.

Las características más destacables de un componente son:

- Es una unidad de *software* reutilizable, con una interfaz bien definida.
- Se distribuye en un único paquete instalable que contiene en sí todo lo necesario para su funcionamiento, con ninguna o muy pocas dependencias de otros componentes o librerías.
- Puede estar implementado en cualquier lenguaje de programación y ser utilizado también para el desarrollo en cualquier lenguaje de programación.
- Es un producto comercial de calidad, realizado por un fabricante especializado. Por supuesto, pueden existir componentes gratuitos.
- Implementación por cualquier lenguaje de programación, especialmente está enfocado a lenguajes de programación orientados a objetos.
- El entorno de desarrollo es ahora más que un simple editor de código. Un entorno de desarrollo orientado a componentes debe facilitar la instalación de componentes y su configuración e integración en una aplicación. El código fuente adicional necesario es mínimo.

6.2 Herramientas de desarrollo de componentes

En este apartado, vamos a hacer una introducción de las herramientas de desarrollo de componentes más conocida en el desarrollo Java: Enterprise Java Beans o EJB.

EJB es una interfaz de aplicaciones que forma parte del estándar de construcción de aplicaciones JEE.

JEE es una plataforma de programación para el desarrollo y la ejecución de *software* con el lenguaje de programación Java.

Con la tecnología de EJB es posible el desarrollo de componentes que luego podremos reutilizar y empaquetarlos para poderlos reutilizar en distintas aplicaciones. El desarrollo de componentes ofrece mayor nivel de funcionalidades.

Para entender mejor qué es el desarrollo en componentes, tenemos que pensar que un componente se podría comparar con una librería, es decir, un componente será un contenedor que tendrá cierta funcionalidad que, cuando se empaquete, se puede añadir a otro proyecto y este usar esos métodos para añadir funcionalidades que antes no tenía. Cuando queremos añadir una librería a un proyecto, lo que hacemos es añadir un .jar. Una vez cargado, este añade funcionalidades extra a nuestro proyecto. El desarrollo de componentes puede asemejarse a eso, para entenderlo de manera simple. Comúnmente, a las clases de este tipo de desarrollo se las llama *beans*.

Podemos destacar tres tipos de Enterprise Java Beans:

1. ***Beans* de sesión**: este tipo de clase contiene la lógica de negocio que puede ser invocada por un cliente local, remoto o de servicio web. La lógica de negocio es lo que se considera el código que desarrolla la función principal de esta clase. Por ejemplo, podemos considerar lógica de negocio los métodos de una clase *bean* que se encargue de realizar las funciones básicas de una calculadora, la gestión de *stock* de una tienda *online* u operaciones de un banco. Podemos diferenciar los beans de sesión en tres tipos:
    - ***Stateless*** (*bean* de sesión sin estado): es un tipo de EJB que no utiliza sesiones para gestionar los datos de los EJB, es decir, no mantiene el estado de la sesión del cliente al realizar las acciones principales. Cuando un cliente invoca los métodos de un *bean stateless*, las variables de instancia del *bean* pueden contener un estado concreto para ese cliente, pero solo durante el periodo que dure la invocación de la instancia de este objeto. Se utiliza para realizar tareas de tipo más rutinario, como,

por ejemplo, operaciones matemáticas, búsquedas u operaciones lógicas más complejas.

- ***Statefull* (*bean* con estado de sesión)**: este tipo se caracteriza por mantener la sesión del cliente en el contenedor EJB y es capaz de mantener el estado de un cliente en más de una solicitud. En un *bean statefull*, las variables de instancia representan el estado de una sesión único de cliente/*bean*. Este estado a menudo se denomina estado de conversación cuando el cliente interactúa con su *bean*. Ejemplos de implementación de este tipo serían una tienda *online*, una gestión de usuarios o la gestión de *stock* de una biblioteca. La información debe gestionarse mediante la sesión para poder ser consultada.
- ***Singleton**:* es el tipo de *bean* que tiene una instancia de sesión única, es decir, se instancia una vez por aplicación y solo existe para ese ciclo de vida de la aplicación. Los *beans* de sesión *singleton* están diseñados para circunstancias en las que una única instancia de Enterprise Bean es compartida y accedida simultáneamente por los clientes.

2. ***Beans* de entidad o *entity:*** este tipo de clase encapsula el estado que puede persistir en la base de datos. Los datos del usuario se pueden guardar en la base de datos a través de los *entity* y más tarde se pueden recuperar de la base de datos en el *bean* de entidad. La clase *bean* contiene lógica relacionada con los datos. Por ejemplo, una implementación sería la utilización de un *bean* de entidad para la lógica de negocio de un banco que quiera reducir el saldo de la cuenta bancaria cuando se producen cargos, o cuando se quieren modificar los datos de un cliente. Esta información está guardada en una base de datos y el objeto que recupera la información será el *bean* de entidad.
3. ***Beans* controlados por mensajes *(message driven beans):*** este tipo de clase tiene un comportamiento parecido al *bean* de sesión, ya que contiene también la lógica de negocio, pero solo se invocan los métodos al pasar un mensaje.

Los beneficios más destacables de la arquitectura de un EJB son:

- Simplificación del desarrollo de las aplicaciones
- Portabilidad del componente: la arquitectura EJB proporciona un modelo simple para generar componentes.
- Independencia: la arquitectura que ofrece esta tecnología permite que el componente que se desarrolle sea apto para cualquier plataforma.
- Altamente personalizable: este tipo de aplicaciones pueden ser customizadas sin acceder a la fuente de código.
- Versátil y escalable: es fácil de utilizar tanto a pequeña como a gran escala. Es fácil de migrar a otros entornos más potentes.

Por otro lado, algunas de las desventajas que debemos tener en cuenta:

- **Tiempo de desarrollo**: el tiempo de desarrollo de un componente mediante EJB es algo más complejo, por lo que puede demorarse.
- **Conocimientos en Java**: los desarrolladores deben tener altos conocimientos en Java para poder desarrollar una aplicación de este tipo.

6.3. Componentes de gestión de información almacenada en ficheros, bases de datos relacionales, objetos relacionales, orientadas a objetos y nativas XML

Como hemos explicado en apartados anteriores, la programación basada en componentes con EJB consiste en la creación de un componente que puede usarse en cualquier aplicación. En este apartado vamos a mostrar cómo crear un componente sencillo con EJB.

Para poder desarrollar nuestra aplicación, crearemos una clase *bean* que contendrá todos los métodos de negocio, es decir, aquellos métodos que se encargarán de realizar las tareas de la aplicación, y la interfaz de negocio.

Para este ejemplo, vamos a realizar un componente que se encargará de realizar cálculos matemáticos. El componente solo realizará los cálculos básicos.

Creación del proyecto en Eclipse

Vamos a mostrar cómo crear un proyecto EJB desde cero en Eclipse. En primer lugar, para crear un proyecto, tendremos que dirigirnos a *File > New > Other* y se abrirá una ventana como esta:

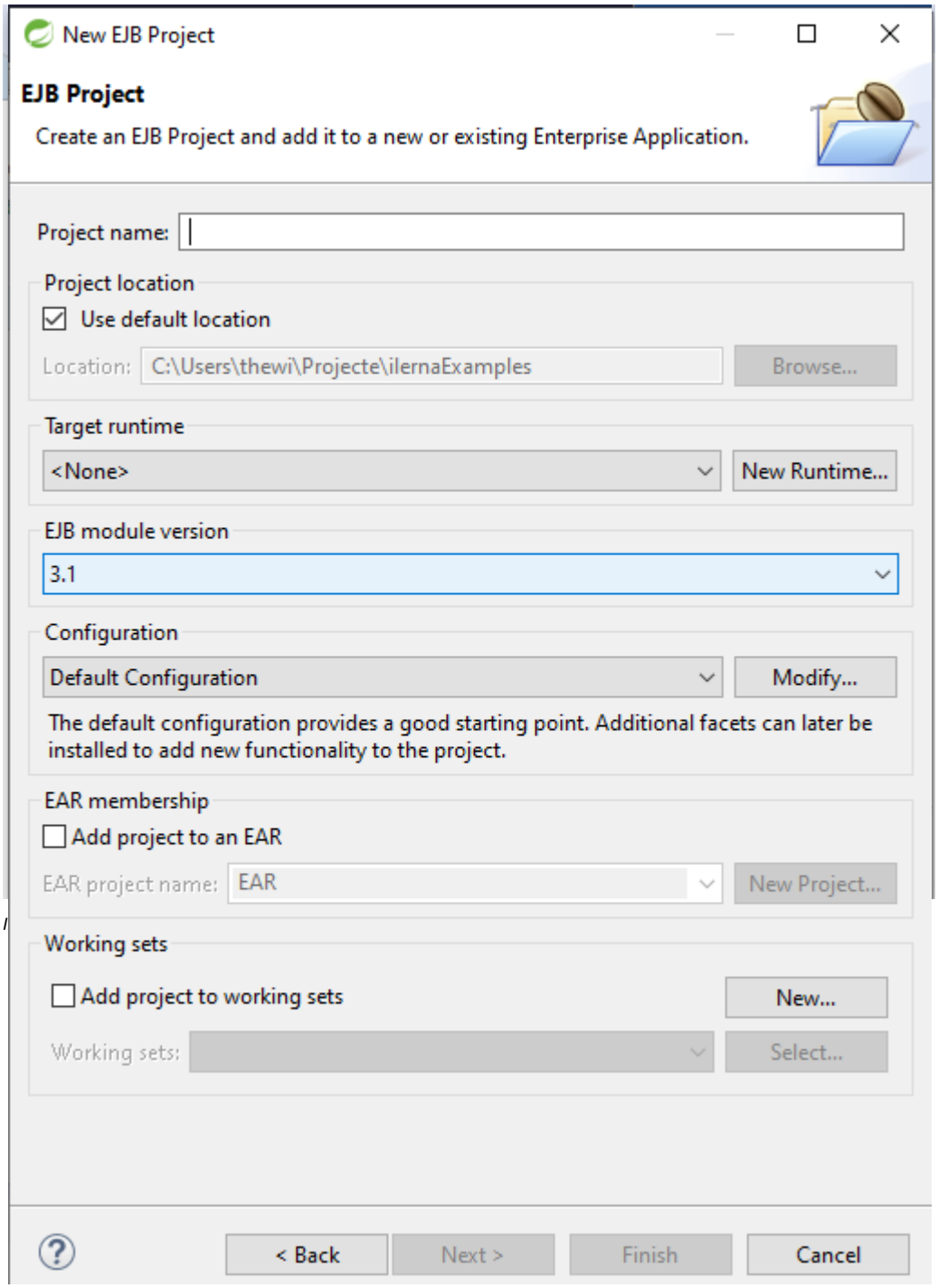


Ilustración 2. Captura de la ventana de configuración de un proyecto EJB.

Buscaremos EJB en el *wizard* y nos saldrán estas opciones. Debemos elegir *EJB Project*. Tendremos que elegir el nombre del proyecto y darle a *Finish*. Para crear los paquetes de nuestra aplicación, tendremos que pulsar con el botón derecho del ratón a nuestro proyecto y a *New*. Elegiremos *Package* y escogeremos el nombre para nuestro paquete. Eclipse se encargará solo de crearlo en la carpeta correspondiente, debería salir algo así:

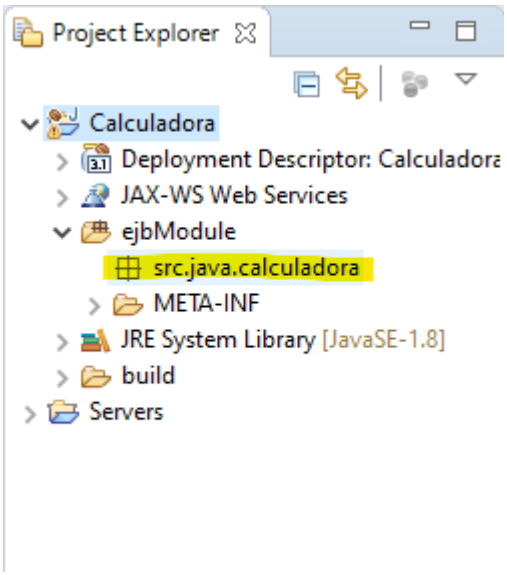


Ilustración 2. Estructura del proyecto EJB.

Para poder desarrollar aplicaciones EJB, antes de empezar tendremos que importar la librería EJB a nuestro proyecto. La podemos descargar directamente desde este enlace:

https://repo1.maven.org/maen2/javax/ejb/ejb-api/3.0/ejb-api-3.0.jar

Creación de la interfaz

Para nuestra calculadora, tendremos que crear una interfaz que recogerá los métodos que vamos a usar. Es importante que cada clase tenga un nombre claro para entender a primera vista a qué se dedica esa clase. Si se trata de una interfaz, le añadiremos al nombre "Negocio", así entenderemos que esa clase contiene toda la lógica de negocio que se tiene que implementar. Así quedaría nuestra primera interfaz de negocio:

```
@Remote
public interface CalculadoraNegocio {
	float suma(float x, float y);
	float resta(float x, float y);
	float multiplicacion(float x, float y);
	float division(float x, float y);
}
```

Como vemos en el código, será una clase pública de tipo interfaz que tiene cuatro métodos para realizar las operaciones, los cuales deben recibir por parámetros argumentos de tipo *float*.

Creación del *bean*

La especificación EJB 3.1 presenta la anotación *@Stateless*, que nos permite crear fácilmente *beans* de sesión sin estado. Como hemos mencionado en el apartado 6.2, un *bean* de sesión sin estado no tiene ningún estado de cliente asociado, pero puede preservar su estado de instancia. El contenedor EJB normalmente crea un conjunto de pocos objetos de *bean* sin estado y los utiliza para procesar la solicitud del cliente.

A continuación, tenemos que crear la clase que implementará la interfaz *CalculadoraNegocio*. El nombre debe seguir la misma terminología, es decir, si a esta le ponemos "Calculadora", como la clase que crearemos implementa la interfaz, deberá llamarse *CalculadoraBean*. El propósito de esta clase será implementar el método de su interfaz. Debemos recordar que, antes de declarar la clase, tenemos que añadir la anotación *@Stateless*.

```
@Stateless
public class CalculadoraBean implements CalculadoraNegocio {

          @Override
          public float suma(float x, float y) {
              return x + y;
          }

          @Override
          public float resta(float x, float y) {
              return x - y;
          }

         @Override
          public float multiplicacion(float x, float y) {
              return x * y;
          }

          @Override
          public float division(float x, float y) {
              return x / y;
          }
}
```

Con la interfaz y la implementación de esta interfaz, ya tenemos nuestro componente creado. Este es un componente sencillo que nos puede ayudar a entender cómo funciona la programación orientada a componentes.

Esta implementación solo está pensada para un *bean stateless* que no tiene necesidad de guardar los datos. Si necesitáramos guardar el resultado de la operación, tendríamos que crear otra capa de negocio, que es la que se encargará de guardar los datos. En primer lugar, crearemos una interfaz *CalculadoraDAO* que definirá el método que se encargará de guardar el resultado a la base de datos:

```
public interface CalculadoraDao {
	public void guardar(float resultado);

}
```

A continuación, tendremos que crear una clase que implemente *CalculadoraDAO*, la llamaremos *CalculadoraDAOImpl*. Esta clase tendrá que tener un método privado que se encargue de realizar la conexión con la base de datos.

```
private Connection connect = null;
	// JDBC driver name and database URL
	static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
	static final String DB_URL = "jdbc:mysql://localhost/ilerna";
	static final String USUARIO = "alumno";
	static final String CONTA = "password";

private Connection conector() throws Exception {
	try {
		// Esto se encarga de cargar el driver de MySQL
		Class.forName(JDBC_DRIVER);
		// Establecemos la conexion
		connect = DriverManager.getConnection(DB_URL + "user=" +
USUARIO + " &password=" + CONTA);

	} catch (Exception e) {
		throw e;
	} finally {
		connect.close();
	}
	return connect;
}
```

Este método nos tiene que sonar del tema 2, es una de las maneras de guardar información a la base de datos: usaremos una conexión con una base de datos MySQL. A continuación, implementaremos el método de guardar:

```
@Override
public void guardar(float resultado) {
	Connection conn = null;
	PreparedStatement stmt = null;
	try {
		// Paso 1: Realizamos la conexion
		conn = conector();
		System.out.println("Nos hemos conectado a la BBDD");
		String sql = "INSERT INTO RESULTADO (id, resultado) VALUES
(22, " + resultado + ")";
		stmt.executeUpdate(sql);

	} catch (SQLException se) {
	// Gestionamos los posibles errores que puedan surgir durante la
ejecucion de la
	// inserciï¿½n
		se.printStackTrace();
	} catch (Exception e) {
		// Gestionamos los posibles errores
		e.printStackTrace();
	} finally {
		// Cerramos la connexion
		try {

			conn.close();
		} catch (SQLException se) {
			System.out.println("No se ha podido cerrar la
conexiÃ³n.");
		}
	}

}
```

Tenemos que tener en cuenta que, antes de guardar un registro, tenemos que tener creada la tabla en la base de datos. La sentencia que se ejecuta en el método que mostramos en el recuadro se guardará en la tabla "resultado", si no existe, dará un *SQLException*. Esta estructura nos debe sonar de temas anteriores. Para ver otras opciones de persistencia, tendremos que revisar el apartado 6.7.

En este método, se encargará de guardar la información del resultado en una tabla. El comportamiento de un componente debe ser parecido tanto si se guardan los datos como si no. Será un bloque de código que realizará una acción y que podrá ser empaquetado para poder ser utilizado en otras aplicaciones.

6.4. Propiedades (simples, indexadas, ligadas y restringidas) y atributos

En este apartado, vamos a explicar en qué consiste una propiedad y un atributo de un *bean* entidad y clasificaremos todos sus tipos.

Para empezar, podemos entender como propiedad:

> Una **propiedad** es un atributo de una clase Java Beans que contiene una serie de atributos con sus métodos de acceso que permiten la modificación de la apariencia o la conducta de una aplicación.

Una propiedad de una clase tiene sus métodos de acceso y de modificación, o una serie de funciones que permitirán la modificación de esa propiedad. Los métodos más conocidos son:

- ***Getter**:* es un método que nos va a permitir obtener la información de una propiedad de una clase.
- ***Setter**:* es el método que se encargará de modificar el valor de la propiedad de una clase.

```
class Empleado {
      private String nombre;

      public getNombre(String nombre){
            this.nombre;
      }
      public setNombre(String nombre){
            this.nombre = nombre;
      }
}
```

GETTER

SETTER

Una propiedad puede clasificarse en cuatro tipos:

6.4.1. Propiedades simples

> Una **propiedad simple** es aquella propiedad que representa un valor único.

Es decir, una propiedad simple es aquella que tiene un *getter* y un *setter*, y el valor no cambiará. Un ejemplo práctico de un *getter* y *setter* es el que hemos visto en el recuadro anterior. Cada propiedad tendrá su método de acceso de lectura y su método de escritura.

6.4.2. Propiedades indexadas

Una **propiedad indexada** es aquella propiedad que representa una lista de valores y que es capaz de devolver todo el listado o el valor que se le indica mediante un índice.

Una propiedad indexada tendrá métodos que permitirán la lectura y la escritura de los elementos individuales de un listado de tipo *array*, y tendrá otros métodos que van a permitir la lectura y escritura de solo un valor que se pasará por parámetro como un índice.

```
public class Persona {
	private int[] objetos = { 1, 2, 3, 4 };

	// método indexado set de la propiedad objetos
	public void setObjetos(int[] nuevoValor) {
		objetos = nuevoValor;
	}
	// método get de la propiedad objetos
	public int[] getObjetos() {
		return objetos;
	}

	// métodos get y set para un elemento de array
	public void setObjetos(int indice, int nuevoValor) {
		objetos[indice] = nuevoValor;
	}

	public int getObjetos(int indice) {
		return objetos[indice];
	}
}
```

Con estos métodos lo que conseguimos es obtener todo el listado o solo uno de los valores de la lista, que es el que corresponde al índice. Por ejemplo, si hacemos uso del método *getObjetos(1)*, el valor que obtendremos del *get* será un 2, porque es el valor que se encuentra en la posición 1.

6.4.3. Propiedades ligadas

Una **propiedad ligada** (también conocida como *bound*) es aquella propiedad que desencadena un evento cuando su valor cambia.

Cuando tenemos una propiedad de este tipo, debemos tener en cuenta las implicaciones que ello conlleva:

- La clase *bean* debe incluir dos métodos: *addPropertyChangeListener()* y *removePropertyChangeListener()* para poder gestionar los *beans* que actúan como *listener* de ese evento.
- Cuando la propiedad ligada cambia de valor, el *bean* envía un evento de tipo *PropertyChangeEvent* que el objeto *listener* recibe (este *listener* será un tipo *PropertyChangeListener*).

Estas clases mencionadas se encuentran en el paquete java.beans. Profundizaremos más en los eventos en el apartado 6.5, pero para que entendamos de qué se trata, vamos a mostrar un ejemplo de cómo definir una propiedad de este tipo.

```
public class Persona {
      private int altura = 90;

      private PropertyChangeSupport soporte = new
      PropertyChangeSupport(this);

      public int getAltura() {
        return altura;
      }

      public void setAltura(int valor
) {
        int oldAltura = altura;
        altura = valor;
        soporte.firePropertyChange("altura", oldAltura, valor);
      }

      public void
addPropertyChangeListener(PropertyChangeListener listener) {
            soporte.addPropertyChangeListener(listener);
      }

    public void
removePropertyChangeListener(PropertyChangeListener listener) {
      soporte.removePropertyChangeListener(listener);
    }
```

Como podemos apreciar, tendremos que declarar una propiedad *PropertyChangeSupport* que se usará en el *set* de la propiedad y que disparará el evento. Los métodos *addPropertyChangeListener()* y *removePropertyChangeListener()* se tendrán que implementar para poder añadir o eliminar el *listener*.

6.4.4. Propiedades restringidas

Una **propiedad restringida** es un tipo de propiedad ligada que es capaz de restringir eventos.

Cuando una propiedad restringida está a punto de cambiar, se realiza una consulta al *listener* sobre ese cambio. Cualquiera de los *listeners* definidos puede comprobar el evento y restringir el cambio. Si un *listener* veta un cambio de una propiedad, ese

atributo no cambia de valor. Una propiedad puede tener dos tipos de *listeners:* el *listener* que veta un evento y el *listener* que captura el evento y enviará la notificación del cambio.

Para poder gestionar las restricciones, disponemos de una clase en el paquete de java.beans que se llama *VetoableChangeSupport* y que facilita la gestión de las propiedades restringidas.

En el ejemplo que encontramos a continuación hemos resaltado en negrita las partes que se añaden para restringir una propiedad. El ejemplo es parecido a una propiedad ligada. Tendremos que añadir una nueva instancia de *VetoableChangeSupport* e implementar el método ***addVetoableChangeListener()*** y ***removeVetoableChangeListener()***. En el *set* de la propiedad, se comprobará si está restringida o no, y se añadirá o no el evento con los métodos nombrados anteriormente.

```
public class Persona {

// miembro de la calse que se usa para guardar el valor de la
propiedad
      private int[] objetos = { 1, 2, 3, 4 };
      private int altura = 90;


    private PropertyChangeSupport soporte = new
PropertyChangeSupport(this);
    private VetoableChangeSupport pRestringida = new
VetoableChangeSupport(this);

    public int getAltura() {
        return altura;
    }

    public void setAltura(int valor) throws PropertyVetoException
{
        int oldAltura = altura;
        pRestringida.fireVetoableChange("altura", oldAltura,
valor);
        altura = valor;

        soporte.firePropertyChange("altura", oldAltura, valor);
    }

    public void addPropertyChangeListener(PropertyChangeListener
listener) {
      soporte.addPropertyChangeListener(listener);
    }

    public void
removePropertyChangeListener(PropertyChangeListener listener) {
      soporte.removePropertyChangeListener(listener);
    }

    public void addVetoableChangeListener(VetoableChangeListener
listener) {
      pRestringida.addVetoableChangeListener(listener);
    }

    public void
removeVetoableChangeListener(VetoableChangeListener listener) {
      pRestringida.removeVetoableChangeListener(listener);
    }

}
```

6.5. Eventos: asociación de acciones a eventos

En este apartado haremos una introducción a los eventos y la asociación de acciones a eventos en Java Beans.

Un **evento** se puede definir como un cambio que se produce durante la ejecución de una aplicación o de un programa.

Para entenderlo mejor, un evento podría ser cualquier tipo de interacción que un usuario realiza con nuestra aplicación, por ejemplo, un clic en un botón, escribir texto en un formulario o seleccionar un elemento de una lista. Como hemos visto en el apartado anterior, una propiedad puede tener un *listener* que desencadene un evento.

La gestión de los eventos se encargará de aprovechar los cambios durante el programa para invocar procedimientos más específicos. También se consideran eventos las interacciones entre clases de nuestro programa, ya que se comunican las unas con las otras para identificar qué ocurre en la aplicación y actuar de una forma u otra.

A partir de la versión 1.1 de JDK, Java dispone de forma estándar un grupo de clases e interfaces que permiten la gestión de eventos en el momento en que se producen para poder asociar procesos condicionados a determinados eventos. Esta gestión de eventos se basa en un patrón de diseño de *software* al que podemos llamar *observer*. Este patrón define un escenario con dos elementos, el que lanza el evento y el que lo recibe (normalmente conocido también como *listener*). El objeto que recibe el evento también ejecuta diferentes eventos, porque es allí donde se producen los cambios y se originan los acontecimientos.

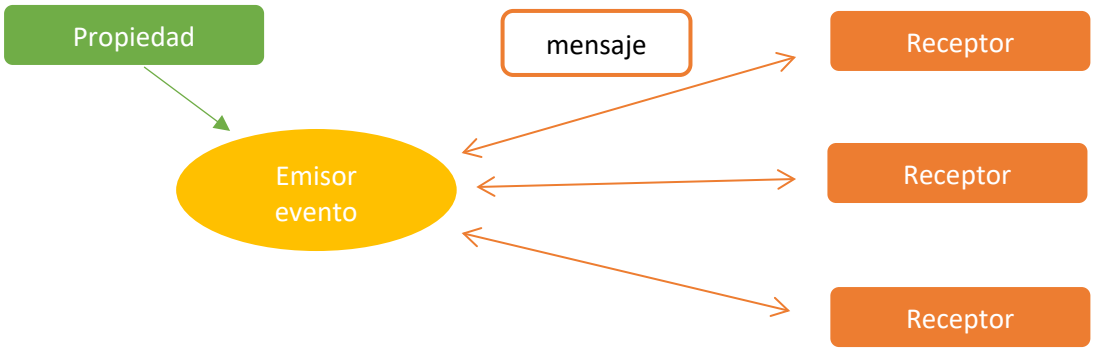


Ilustración 3. Esquema de funcionamiento de eventos y listeners.

Para poder enviar un evento, el receptor deberá indicar al emisor que le avise cuando haya un cambio. El receptor permanecerá pasivo hasta que sea avisado, momento en

que activará los procesos que sean necesarios. En la práctica, este patrón necesita un tercer elemento: el evento. Se trata del elemento encargado de notificar a los receptores qué cambio se ha producido. Veamos ahora cómo podemos implementar con Java un sistema de estas características.

En Java, para gestionar eventos, podemos encontrar la clase *EventObject* que se encuentra en el paquete java.util. La función principal de esta clase es esperar hasta recibir en su constructor la fuente de datos del evento, y de esta manera el receptor podrá analizar los cambios sufridos y actuar consecuentemente.

Si en cada notificación necesitamos enviar otros datos complementarios, podemos implementar clases derivadas de *EventObject* que reciban los datos necesarios durante la instanciación. Las más destacables son:

- *ChangeEvent*.
- *ConnectionEvent*.
- *DropTargetEvent*.
- *Notification*.
- *PropertyChangeEvent*.
- *StatementEvent*.
- *TableModelEvent*.

En la documentación de Java podremos encontrar todos los tipos de eventos que se permiten y que pueden adaptarse a las necesidades de nuestra aplicación, los mencionados solo son unos cuantos ejemplos que tener en cuenta.

https://docs.oracle.com/javase/7/docs/api/java/util/EventObject.html

La fuente de eventos, o clase de objetos observables, será una clase de Java con la particularidad de que deberá tener una operación para asociar receptores y otra para eliminarlos. Necesitaremos también definir una interfaz con un único método que usaremos para realizar las notificaciones. Cualquier clase que tenga que hacer de receptor de la fuente de eventos tendrá que implementar la interfaz para conseguir la compatibilidad.

Para poder definir una jerarquía diversa, Java dispone de una interfaz sin métodos llamada *EventListener* que se encuentra en el paquete java.util. Aunque no es obligatorio, es aconsejable que todos los receptores pertenezcan a la misma jerarquía y, por tanto, se deriven.

Por ejemplo, si creamos una clase que recibirá un evento, tendremos que declararla de esta manera:

```
public class EjemploEvento extends EventObject {

	public EjemploEvento(Object source) {
		super(source);
	}
}
```

Nuestra clase *EjemploEvento* extenderá de *EventObject*, y tendrá que tener un constructor como el que vemos en el ejemplo. A través de este constructor, le pasaremos los datos al receptor.

Para crear el receptor, crearemos una interfaz que extienda de *EventListener*. Vamos a ver un ejemplo sencillo:

```
public interface EjemploListener extends EventListener{
	public void ejemploPracticoEvento(EjemploEvento event);
}
```

La gestión de eventos es una opción útil a la hora de trabajar con componentes, porque nos va a permitir que cada componente puede interactuar con otro sin necesidad de saber cómo está construido el código ni cuáles serán los componentes que van a interactuar.

6.6. Introspección. Reflexión

La **introspección** es un proceso automático que analiza los patrones de diseño de un *bean* (es decir, una clase Java) para revelar propiedades, eventos y métodos. Este proceso controla la publicación y el descubrimiento de operaciones y propiedades del *bean*. Este proceso utiliza la reflexión.

En este apartado, explicaremos en qué consisten la introspección y la reflexión enfocadas en la programación orientada a objetos. En Java, cuando construimos una aplicación, intentamos transformar objetos de la vida real en clases Java para poder transformar una necesidad en una aplicación. Con la introspección pasa algo parecido: la introspección es una capacidad de un programa de examinarse a sí mismo viendo las

capacidades de los objetos creados en **tiempo de ejecución**, sus propiedades y sus métodos para modificarlos.

> Un ***bean*** es una clase de Java que sigue los estándares definidos por JavaBeans. Es un modelo de componentes, una clase que tiene sus propiedades, sus constructores y sus *getter* y *setter*.

Por otro lado, tenemos la **reflexión**. Podemos entender como reflexión:

> Es la capacidad que tiene un programa para observar y modificar su estructura de manera dinámica, es decir, durante el proceso de ejecución de un programa.

Gracias a ambas características, un programa es capaz de ver la naturaleza de las clases, atributos y métodos para poder modificarlos en tiempo de ejecución.

Para poder realizar las tareas de introspección y reflexión, usaremos estas clases: *Class*, *Method*, *Constructor* y *Field*, todas ellas son subclases de la clase *Object* que se encuentra en el paquete java.lang. Eso significa que todas las clases heredarán los métodos de *Object*. El método más importante que utilizarán todas las clases a lo largo del tema es:

Tipo	Método	Descripción
Class<?>	getClass()	Devuelve el tipo de clase a la que pertenece la clase que se está consultando en tiempo de ejecución.

Las clases que nos van a permitir la introspección en Java son:

6.6.1 *Class*

La clase *Class* representará las clases y las interfaces que se estén ejecutando en nuestra aplicación Java. Nos va a permitir determinar a qué clase pertenece un objeto determinado dentro del código. Los métodos de *Class* se usan en otras clases que veremos en siguientes páginas, por eso, debemos prestar atención a estos métodos y memorizar los más importantes:

Tipo	Método	Descripción
Method []	getDeclaredMethods()	Devuelve una lista en forma de *array* de tipo *Method* con todos los métodos de esa clase.
Method	getMethod(String nombre)	Devuelve los métodos que pertenecen a una clase.
Method[]	getMethods()	Devuelve todos los métodos públicos de una clase en un *array* de tipo *Method*.
	getModifiers()	Nos devuelve los modificadores de acceso de la clase.
boolean	isInstance(Object obj)	Determina si el objeto especificado es una instancia de la clase que representa.
boolean	isInterface()	Este método comprueba si la clase es una interfaz.
Package	getPackage()	Devuelve el paquete en el que se encuentra una clase.
T	newInstance()	Crea una nueva instancia de la clase que representa.
String	toString()	Convierte un objeto en *string*.

Tipo	Método	Descripción
static Class<?>	forName(String nombreClase)	Devuelve el objeto de clase asociado con la clase o interfaz con el nombre de cadena dado.
Constructor<T>	getConstructor(Class<?>… tipoP)	Devuelve un objeto *Constructor* con el constructor público que le pasamos por parámetro.
Constructor<T>[ ]	getConstructors()	Devuelve un *array* de tipo *Constructor* con todos los constructores públicos de esa clase.
Constructor<T>	getDeclaredConstructor(Class<?>... tipoParametro)	Devuelve un objeto *Constructor* con el constructor del nombre que le pasamos por parámetro, ya sea público o privado.
Constructor<T>[ ]	getDeclaredConstructors()	Devuelve un *array* de tipo *Constructor* con todos los constructores de esa clase.
String	getName()	Devuelve el nombre de la clase a la que representa.
Array	getField()	Devuelve los campos de una clase.
Field	getDeclaredField(String nombre)	Devuelve la información de un atributo que le pasamos por parámetro.
Field[]	getDeclaredFields()	Devuelve la información de todos los atributos de una clase.

Method	getDeclaredMethod(String name,Class<?>... parameterTypes)	Devuelve la información de un método indicado a través de un *string* por parámetro.

Para ver cómo funciona esta clase y cómo nos puede ser de utilidad, vamos a ver un ejemplo usando algunos de los métodos descritos en el cuadro anterior. Partiremos de una clase simple a la que nombraremos "Persona" y que usaremos para los ejemplos.

```
class Persona {
	private String nombre;
	public Persona(String nombre){
		this.nombre = nombre;
	}
	public String getNombre(){
		return nombre;
	}
	public String getNombreApellido(String apellido){
		return  this.nombre + “ ” + apellido;
	}
}
```

Como vemos, esta clase tendrá su constructor y sus propiedades, cada propiedad es privada y los constructores son públicos.

Lo que nos permite la introspección es tener información sobre las clases, métodos y propiedades de las clases Java en tiempo de ejecución. Cuando ejecutamos un programa, la máquina virtual de Java va creando instancias de *Class* para cada tipo de objeto. Esta *Class* incluye diferentes métodos que nos van a permitir examinar propiedades e información de ese objeto.

Si por ejemplo queremos averiguar a qué clase pertenece un objeto, usaremos el método *getClass()* de esta forma:

```
Persona persona = new Persona("Laura");
System.out.println(persona.getClass());
//almacenamos la información de la clase Laura en la
//clase Class.
Class cl = persona.getClass();
```

Podemos hacerlo de dos maneras: directamente usando el objeto *persona.getClass()* e imprimiéndolo en la consola o, por otro lado, podemos crear una *Class* para que almacene la información de *persona.getClass()*.

Si, por lo contrario, no sabemos el nombre de una clase en tiempo de compilación, podemos averiguar su clase en tiempo de ejecución a partir de un *string* que contenga el nombre de la clase. En este *string* se puede incluir el paquete al que pertenece junto con el nombre de la clase con el método *forName()*. Vamos a ver un ejemplo para entender su implementación.

```
Class clPersona = Class.forName("com.test.introspeccion.Persona");
```

Si al método *forName* le pasamos por parámetro el nombre del paquete y el nombre de la clase, rellenará la *Class*. Este método también nos puede servir para cambiar el nombre de una clase en el momento de la ejecución de la aplicación. Lo podríamos hacer de este modo:

```
String nombreClase ="introspeccion.Empleado";
try{
      //Posibilidad de cambiar el nombre de una clase en tiempo de
ejecución.
      cl = Class.forName(nombreClase);
      System.out.print(persona.getName());
} catch(ClassNotFoundException e){

}
```

Como vemos en el ejemplo, declararemos un nuevo *string* "nombreClase" al cual le asignaremos el nombre nuevo. Con el método *forName()* podremos cambiar el nombre de la clase declarada durante el proceso de ejecución. Cuando se ejecute nuestra pequeña aplicación, podremos ver el nombre nuevo por la consola usando el método *getName()*. Cuando realicemos un cambio de este tipo, tendremos que englobar el código en un *try-catch* por si el programa no encuentra la clase y se produce un error.

6.6.2 *Field*

La clase *Field* nos proporcionará información sobre un atributo de una clase o de una interfaz. Nos da información sobre el tipo de atributo y el acceso que se usa. Los métodos más destacables de la clase *Field* son:

Tipo	Método	Descripción

boolean	equals(Object objeto)	Compara un objeto con el que se pasa por parámetro.
boolean	getBoolean(Object objeto)	Obtiene el valor de un campo booleano estático o de instancia.
char	getChar(Object objeto)	Obtiene el valor de un atributo estático o de instancia de tipo *char* o de otro tipo primitivo convertible a tipo *char* mediante una conversión de ampliación.
double	getDouble()	Obtiene el valor de un atributo estático o de instancia de tipo *double* o de otro tipo primitivo convertible a tipo *double* mediante una conversión de ampliación.
float	getFloat()	Obtiene el valor de un atributo estático o de instancia de tipo *float* o de otro tipo primitivo convertible a tipo *float* mediante una conversión de ampliación.
int	getInt()	Obtiene el valor de un atributo estático o de instancia de tipo *int* o de otro tipo primitivo convertible a tipo *int* mediante una conversión de ampliación.
long	getLong()	Obtiene el valor de un atributo estático o de instancia de tipo *long* o de otro tipo primitivo convertible a tipo *long* mediante una conversión de ampliación.
int	getModifiers()	Devuelve los modificadores que Java usa para representar el objeto.
String	getName()	Devuelve el nombre del atributo.
Class<?>	getType()	Devuelve un objeto con la clase que identifica al atributo que se consulta.
String	toString()	Convierte la clase en uso en un *string*.

Para acceder a los atributos públicos de una clase podemos hacerlo de dos maneras. Si conocemos el nombre del atributo de la clase, usaremos este método: *getField()*. Para este ejemplo usaremos una clase nueva:

```
class Empleado {
private String nombre;
public String apellido;

	public Empleado(String apellido){
		this.apellido = apellido;
	}

	public String getApellido(){
		return apellido;
	}
}
```

```
Empleado empl = new Empleado("Gonzalez");
Field emplFields = empl.getField("apellido");
```

Como vemos en el ejemplo, a través del método *getFields()* y pasándole por parámetro el nombre del atributo, podremos obtener toda la información de ese campo, ya sea el tipo de datos o el modo de acceso, entre otros. Debemos tener en cuenta que este método solo será útil si el atributo se encuentra en modo público. Si no existiera ningún campo "apellido", el programa daría un error de tipo *NoSuchFieldException*.

En cambio, si no sabemos los nombres de los atributos de nuestra clase, podemos obtener la información de esta manera:

```
Empleado empl2 = new Empleado("Martinez");
Field[] atributos = empl2.getFields();
```

El método *getFields()* nos devolverá un *array* de tipo *Field* por cada uno de los atributos públicos que encuentre en la clase. En este caso, solo encontrará uno y nuestro *array* solo tendrá un valor.

Para acceder a los atributos privados de nuestra clase usaremos el método *getDeclaredField()*, que necesita que pasemos por parámetro un *string* con el nombre del atributo si sabemos cuál es. Si, por el contrario, no sabemos el nombre de los atributos, usaremos *getDeclaredFields()*. Vamos a ver cómo implementarlo:

```
Empleado empl3 = new Empleado("Lopez");
Field empl3Fields = empl3.getDeclaredField("nombre");

Empleado empl4 = new Empleado("Gomez");
Field[] empl4Fields = empl4.getDeclaredFields();
```

Para manipular los atributos obtenidos en la clase *Field*, tenemos algunos métodos que nos van a permitir obtener más información: con el método *getName()* podremos obtener el nombre del atributo, y con *getType()* nos dirá el tipo de dato usado. También podremos usar los *sets* correspondientes. Para tener más información de los métodos de la clase *Field*, es recomendable repasar la tabla anterior con todos los métodos detallados.

```
Empleado empleado = new Empleado("Gomez");
Field empleadoField= empleado.getDeclaredField("nombre");

String nombreAtributo = empleadoField.getName();
Object tipoAtributo = empleadoField.getType();
```

6.6.3 *Constructor*

La clase *Constructor* nos va a permitir obtener información sobre cuántos constructores y de qué tipo tiene una clase. Los métodos que más vamos a usar de esta clase son los mismos que se usan en la clase *Field*.

Tipo	Método	Descripción
boolean	Equals(Object obj)	Compara un objeto con el que se pasa por parámetro.
Object	getDefaultValue()	Devuelve el valor predeterminado para el miembro de anotación representado por esta instancia de *Method*.
Type[]	getGenericParameterTypes()	Devuelve una matriz de objetos *Type* que representan los tipos de parámetros formales, en orden de declaración, del método representado por este objeto *Method*.

Type	getGenericReturnType()	Devuelve un objeto *Type* que representa el tipo de retorno formal del método representado por este objeto *Method*.
int	**getModifiers()**	Devuelve los modificadores del lenguaje Java para el método representado por este objeto *Method*, como un entero.
String	getName()	Devuelve el nombre del método representado por este objeto *Method*, como un *String*.
Class<?>[]	getParameterTypes()	Devuelve una matriz de objetos de clase que representan los tipos de parámetros formales, en orden de declaración, del método representado por este objeto *Method*.
Class<?>[]	getReturnType()	Devuelve un objeto *Class* que representa el tipo de retorno formal del método representado por este objeto *Method*.
TypeVariable<Method>[]	getTypeParameters()	Devuelve una matriz de objetos *TypeVariable* que representan las variables de tipo declaradas por la declaración genérica representada por este objeto *GenericDeclaration*, en orden de declaración.

Para poder saber cuántos constructores tiene una clase en concreto, tendremos que utilizar el método *getConstructors()*, e implementarlo se realiza de esta manera tan simple:

```
Persona persona = new Persona("Laura");
Class personaClass = persona.getClass();
Constructor[] constructoresPersona =
personaClass.getConstructors();
```

En este ejemplo, definimos un objeto nuevo "persona" de la clase "Persona". Seguidamente, asignamos los datos de la clase a un nuevo objeto *Class*. Como vemos, para obtener los constructores, declaramos un *array* de *Constructor* para que guarde todos los métodos que tenga esa clase. Si nuestra clase no tiene *constructor*, el programa lanzaría una excepción de tipo *NoSuchMethodException*.

Si, por ejemplo, queremos saber los parámetros de un *constructor*, con la reflexión también podemos obtener los datos. Solo tendremos que crear un *array* de tipo *Class* y usar el método *getParameterTypes()*. Vamos a ver un ejemplo de su utilización.

```
Persona persona = new Persona("Laura");
Class personaClass = persona.getClass();
Constructor[] constructoresPersona =
personaClass.getConstructors();
Class[] constructoresPersona =
constructoresPersona.getParameterTypes();
```

A través de la clase *Constructor*, también podemos crear nuevas instancias de un objeto mediante el método *newInstance()*, pasándole los parámetros al constructor si hace falta. Podemos ver cómo funciona en este ejemplo:

```
Persona persona = new Persona("Laura");
Class personaClass = persona.getClass();
Constructor[] constructoresPersona =
personaClass.getConstructors();
Persona persona2 = constructoresPersona.newInstance("Alberto");
```

6.6.4 *Method*

La clase *Method* nos va a permitir obtener información sobre los métodos públicos de una clase o de una interfaz.

Tipo	Método	Descripción
boolean	Equals(Object obj)	Compara un objeto con el que se pasa por parámetro.
Object	getDefaultValue()	Devuelve el valor predeterminado para el miembro de anotación representado por esta instancia de *Method*.

Type[]	getGenericParameterTypes()	Devuelve una matriz de objetos *Type* que representan los tipos de parámetros formales, en orden de declaración, del método representado por este objeto *Method*.
Type	getGenericReturnType()	Devuelve un objeto *Type* que representa el tipo de retorno formal del método representado por este objeto *Method*.
int	**getModifiers()**	Devuelve los modificadores del lenguaje Java para el método representado por este objeto *Method*, como un entero.
String	getName()	Devuelve el nombre del método representado por este objeto *Method*, como un *String*.
Class<?>[]	getParameterTypes()	Devuelve una matriz de objetos de clase que representan los tipos de parámetros formales, en orden de declaración, del método representado por este objeto *Method*.
Class<?>[]	getReturnType()	Devuelve un objeto *Class* que representa el tipo de retorno formal del método representado por este objeto *Method*.
TypeVariable<Method>[]	getTypeParameters()	Devuelve una matriz de objetos *TypeVariable* que representan las variables de tipo declaradas por la declaración genérica representada por este objeto *GenericDeclaration*, en orden de declaración.

Tal y como hemos visto en las clases anteriores, podemos obtener los métodos públicos pasándole al método por parámetro el nombre del método si tenemos constancia del nombre, u obtener todos los métodos públicos en un *array*. Pero también podemos hacer lo mismo tanto por los métodos privados como públicos, sabiendo o no el nombre del método. Vamos a ver un ejemplo rápido usando la misma clase "Persona".

```
Persona persona = new Persona("Laura");
Class personaClass = persona.getClass();
Method[] metodoPersona = personaClass.getMethods();
```

Para usar el método específico necesitamos saber el nombre del método, pero también si necesita parámetros y el orden en que se utilizan. Así, para obtener el método *getNombreApellido()* de la clase "Persona", escribiríamos:

```
Persona persona = new Persona("Laura");
Class personaClass = persona.getClass();
Method metodoEspPersona =
personaClass.getMethod("getNombreApellido", new
Class[]{String.class});
```

Tal y como hemos visto con los atributos, estos métodos solo nos devuelven la información de los métodos públicos de una clase. Para poder acceder a los métodos privados tendremos que usar los métodos *getDeclaredMethod()* y *getDeclaredMethods()*.

Una vez almacenada la información del método en la clase *Method*, podremos empezar a manipular la información. Con este tipo de clase podemos obtener el nombre del método y el tipo de dato que devuelve, usando métodos análogos a los que usamos en los atributos:

```
Persona persona = new Persona("Laura");
Class personaClass = persona.getClass();
Method metodoEspPersona =
personaClass.getMethod("getNombreApellido", new
Class[]{String.class});
String nombreMetodo = metodoEspPersona.getName();
Object tipoMetodo = metodoEspPersona.getReturnType();
```

6.7 Persistencia del componente

Como bien sabemos, la persistencia es un medio que utiliza cualquier aplicación para recuperar datos y almacenarlos en la base de datos. La persistencia es un proceso muy importante en las aplicaciones que utilizan bases de datos para almacenar su información.

En este apartado profundizaremos en la API de persistencia de Java (JPA).

> JPA es una librería de Sun Microsystems y un ORM para Java que gestiona datos relacionales y facilita su persistencia en la base de datos y la correlación relacional de objetos.

A través de JPA, el desarrollador puede asignar, almacenar, actualizar y recuperar datos de bases de datos relacionales a objetos Java y viceversa. JPA se puede utilizar en aplicaciones Java-EE y en contenedores EJB.

Este tipo de interfaz se caracteriza por poder relacionarse con cualquier ORM que utilicemos para persistir los datos, como, por ejemplo, Hibernate. Este tipo de librería define la relación entre objetos internamente.

Para poder establecer una relación entre las clases que tengamos en nuestra aplicación y las tablas de la base de datos, JPA crea un estándar de relación mediante XML o anotaciones. Seguramente, este estándar nos sonará de Hibernate, pues actúa de una manera similar. Para simplificar las tareas de persistencia, se utiliza otra API llamada *EntityManager*, que se caracteriza por:

- Facilitar la actualización, obtención, borrado o inserción de objetos en la base de datos.
- Gestionar las operaciones sin necesidad de código JDBC o SQL.
- Ejecutar consultas con lenguaje SQL.

JPA se compone de cinco clases importantes que nos servirán para poder gestionar la persistencia de nuestra aplicación. Estas clases son:

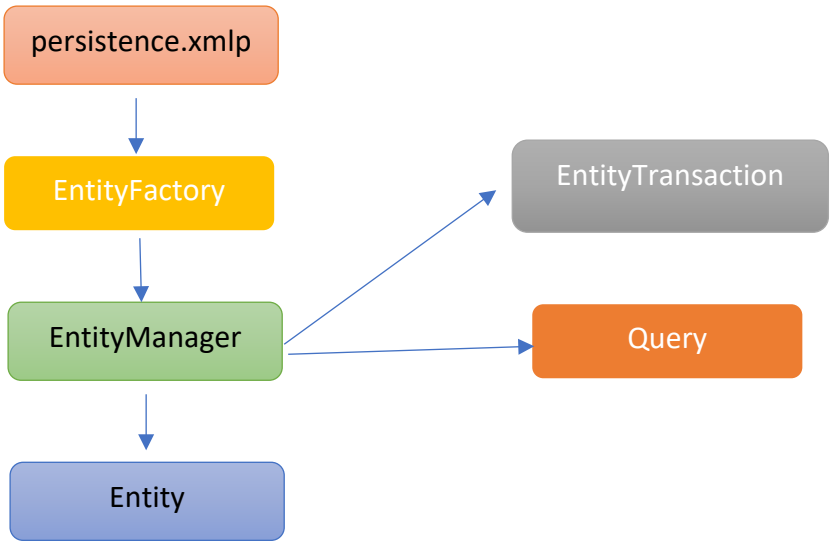


Ilustración 4. Esquema de la estructura de persistencia.

- **Persistence.xml**: es un fichero de configuración que podremos encontrar en nuestra aplicación dentro de la carpeta META-INF. Este archivo se encargará de conectar la base de datos con el conjunto de entidades que la aplicación necesita gestionar. Aparte, contendrá todas las clases que gestionará definidas en el XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="TestPersistence" transaction-
type="RESOURCE_LOCAL">
        <class>
<!-- Entity Manager Class Name -->
      </class>
        <properties>
            <property name="javax.persistence.jdbc.driver"
value="Driver de la base de datos" />
            <property name="javax.persistence.jdbc.url"
value="Url_base_datos" />
            <property name="javax.persistence.jdbc.user"
value="usuario" />
            <property name="javax.persistence.jdbc.password"
value="contraseña" />
        </properties>
    </persistence-unit>
```

Como podemos ver, el fichero sigue una estructura XML, y se tiene que guardar en la carpeta META-INF de nuestro proyecto. Este fichero es muy importante, ya que realiza la conexión con nuestra base de datos e identifica las entidades que se necesita guardar en la base de datos. Estas entidades se definen dentro del tag *<class>* y se establece poniendo el paquete en que se encuentra la clase y su nombre. Cuando tengamos definidos los *entity*, deberemos definir las propiedades de la conexión con la base de datos. Para ello, utilizaremos el *tag <properties>*, y dentro de él un *<property>* por cada valor: el *driver*, la URL de la base de datos, el usuario y la contraseña. Si recordamos cómo configuramos Hibernate, este proceso puede parecerse al que realizamos a lo largo del tema 3.

- ***EntityManagerFactory**:* es una clase que nos facilita la creación de un *EntityManager* para poder manejar las interacciones con la base de datos. Representa un contexto de persistencia. Podemos entender como contexto de persistencia las instancias activas que una aplicación está gestionando en el momento de ejecución.
- ***EntityManager**:* es la interfaz resultante que se encarga de la gestión entre *entitys*, es decir, entre todos los recursos de entidad que está utilizando una aplicación. Será el que ejecutará las acciones entre las entidades y la base de datos. Todos los objetos que se manipulen por un *EntityManager*, primero pasarán por un *PersistenceContext*, otra clase de la que hablaremos a continuación. De esta interfaz podemos destacar los siguientes métodos, que pueden resultarnos útiles para realizar operaciones con la base de datos:

Tipo	Método	Descripción
void	clear()	Se encarga de limpiar el contexto de persistencia.
void	close()	Se encarga de cerrar la instancia.
Query	CreateNamedQuery(String query)	Crea una instancia *Query* con la sentencia que se le pase por parámetro.
Query	createQuery(CriteriaDelete deleteQuery)	Crea una instancia de *Query* para ejecutar una sentencia para eliminar un registro.
<T> T	merge(T entity)	Combina el estado de la entidad dada por parámetro de la persistencia actual.
void	persist(Object entity)	Se encarga de persistir el objeto que se le pasa por parámetro.
void	remove(Object entity)	Elimina la instancia de la entidad indicada por parámetro.

- ***PersistenceContext**:* es la clase que se utiliza cuando un *EntityManager* invoca uno de sus métodos para realizar alguna de las acciones contra la base de datos, ya sea un *persist*, un *merge* o un *remove*, entre otras.
- ***Entity**:* una clase entity será la que represente el objeto que se tiene que persistir en la base de datos. Este objeto será el que un EntityManager gestione durante la operación.
- ***EntityTransaction**:* es una interfaz que se encarga de gestionar las transacciones de la base de datos. Cada *EntityManager* tiene una única instancia de *EntityTransaction* adjunta que está disponible a través del método *getTransaction()*. Como hemos visto en otros temas, una transacción empezará con el método *begin()* y finalizará con el método *commit()*. Si se produce un error, se llamará al método *rollback()*, como en otros métodos de gestión de persistencia.
- ***Query**:* es la clase que se encargará de realizar las consultas a la base de datos y todas las operaciones que tengan que ver con ella. Los métodos más importantes de esta clase son:

Tipo	Método	Descripción
int	executeUpdate()	Ejecuta una sentencia *update* o *delete*.
List	getResultList()	Este método se encarga de devolver una lista de objetos de una sentencia *select*.
Object	getSingleResult()	Este método se encarga de ejecutar una sentencia *select* que retorna un único resultado.

Para entender mejor cómo funcionan todas estas clases, vamos a realizar un ejemplo sencillo de persistencia.

```
public class JPAConexion {
    private static final String PERSISTENCE_UNIT_NAME =
"nombre_definido_fichero_persistance";
    private static EntityManagerFactory factory;

    public static void main(String[] args) {
        factory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
        EntityManager em = factory.createEntityManager();
        // read the existing entries and write to console
        Query q = em.createQuery("select t from Todo t");
        List<Todo> todoList = q.getResultList();
        for (Todo todo : todoList) {
            System.out.println(todo);
        }
        System.out.println("Size: " + todoList.size());

        // comienzo de la transaccion.
        em.getTransaction().begin();
        Todo todo = new Todo();
        todo.setSummary("This is a test");
        todo.setDescription("This is a test");
        em.persist(todo);
        em.getTransaction().commit();

        em.close();
    }
}
```

Para poder establecer una conexión con la base de datos, en primer lugar tenemos que crear un *EntityManagerFactory:* con la llamada al método *createEntityManagerFactory()* y el nombre que hemos definido en el fichero persistance.xml crearemos la instancia de este objeto.

A continuación, necesitaremos crear un nuevo objeto *EntityManager* y una nueva instancia gracias al objeto *EntityManagerFactory* y la llamada al método *createEntityManager()*. La clase EntityManager será una clase importante en la ejecución de consultas y operaciones en la base de datos.

A continuación, tenemos que definir un nuevo objeto de tipo *Query*, que será el que utilizaremos para almacenar los datos de las consultas que ejecutemos. Mediante el *EntityManager* y el método *createQuery()* podremos realizar todo tipo de consultas.

6.8 Herramientas para desarrollo de componentes no visuales

Este tipo de herramientas también son popularmente conocidas como IDE (entorno de desarrollo integrado). ¿Qué podemos entender por IDE?

> Un IDE es un programa enfocado a aquellos profesionales que se encargan de desarrollar aplicaciones y que facilita la tarea de desarrollo con sus funcionalidades.

Un *software* de este tipo suele traer incorporadas una serie de herramientas, como un editor, un compilador, una consola y un depurador. Estas herramientas facilitarán las tareas de desarrollo y depuración del código de una aplicación. Muchas de ellas pueden soportar *plugins* para complementar más aún sus funcionalidades.

Para Java, podemos encontrar multitud de IDE que pueden sernos útiles. En este apartado vamos a mostraros los tres más importantes:

1. **Eclipse**: el IDE más popular actualmente, es un *software* de código abierto y muy extendido en programadores. Este programa fue creado por IBM, aunque ahora sea de código abierto, y tiene una gran comunidad detrás que proporciona una gran cantidad tanto de soporte como de diferentes *plugins*. Eclipse se caracteriza por ser un IDE multiplataforma (soporta Windows, Linux y Mac) y está altamente actualizado, pues recibe varias actualizaciones al año. Aparte de ser multiplataforma, también es multilenguaje, ya que soporta gran cantidad de lenguajes para el desarrollo tanto de aplicaciones web como de escritorio o de móvil usando C, C++, Java, JSP, Perl, Ruby, Python o PHP, entre otros.
   Entre sus mejores cualidades podemos destacar la gran capacidad de *plugins* que podemos añadir, las recomendaciones a medida que se programa, ya sean validaciones o sugerencias, y su gran capacidad de depuración del código. Tiene una de las mejores herramientas de *debug*. Como punto negativo, puede llegar a consumir muchos recursos si la aplicación que se desarrolla es de gran tamaño.
2. **NetBeans**: es otra herramienta de código abierto que podemos utilizar para el desarrollo de aplicaciones. Su principal característica es que permite el desarrollo de aplicaciones por módulos, es decir, el desarrollo de componentes. Para el propósito de este tema, este es un IDE ideal para poder desarrollar nuestros componentes, ya que podremos reutilizar el código para diferentes proyectos sin complicaciones. Aparte de Java, este IDE también es capaz de soportar otros lenguajes de programación, como C, C++, PHP y HTML5. El único

inconveniente de Netbeans frente Eclipse es que no cuenta con una cantidad tan amplia de *plugins*.

3. **IntelliJ Idea**: es un IDE desarrollado por Jet Brains y cuenta con dos versiones: la de pago y la de descarga libre. También se caracteriza por ser multiplataforma y soporta gran cantidad de lenguajes, como Java, Rust, Perl, XML o Kotlin. Estos son los lenguajes que soporta la versión gratuita, la versión de pago también incluye PHP, HTML, CSS y Ruby, entre otros. Ofrece la posibilidad de integración con GIT, lo que nos permitirá gestionar el código de manera eficiente. Como aspecto negativo, podemos destacar las limitaciones de la licencia libre y la escasez de *plugins* respecto de Eclipse.

Estas son las herramientas más destacables para el desarrollo con Java. En todos los apartados y los temas que hemos estado enseñando a lo largo del curso nos hemos centrado en Eclipse, ya que es la mejor opción para el desarrollo Java ahora mismo. No obstante, esta es una opinión subjetiva, por lo que es bueno que tengáis conocimiento de todas las posibilidades que hay en el mercado y que escojáis la que más os guste y se adecúe a vuestras necesidades.

6.9 Empaquetado de componentes

En este apartado, explicaremos en qué consiste el empaquetado de componentes en JavaBeans. Una de las características básicas de cualquier componente es la unidad. Para poder distribuir e integrar en diferentes aplicaciones los componentes, es necesario poder empaquetarlos para poder copiarlos fácilmente donde se requiera. Un componente se divide en recursos, y podemos diferenciar dos tipos de recursos:

- Los recursos que conformarán el componente funcional: serían los ejecutables, las imágenes, el diseño de las interfaces graficas, etcétera. Acostumbran a empaquetarse juntos en un único fichero para facilitar su distribución. El código ejecutable se suele organizar en ficheros de bibliotecas dinámicas. A menudo, es posible incorporar otros recursos, como imágenes o iconos, dentro de la misma biblioteca para poder reducir el número de ficheros implicados.
- Los recursos de instalación y configuración necesarios para adaptar el componente a una aplicación concreta: en cambio, los recursos de configuración suelen mantenerse en ficheros independientes para facilitar su modificación.

Cuando los componentes se implementan con otros lenguajes interpretados o pseudocompilados como Java, también habrá que empaquetar de alguna manera el componente, pero depende de las utilidades de que dispone cada lenguaje.

El lenguaje Java no crea un único archivo binario ejecutable, sino que compila por separado cada clase y las clases compiladas se vinculan durante el proceso de ejecución en la propia máquina virtual. Por esta razón, Java no necesita crear un único archivo ejecutable.

Sin embargo, para mantener el código de los componentes agrupados, Java suele empaquetar todas las clases compiladas en un único fichero de tipo JAR. Se trata de un archivo de formato .zip constituido al menos por las clases compiladas y por un fichero descriptivo llamado manifest.mf que se ubica en la carpeta interna del archivo JAR que se llama "meta-inf".

Un archivo manifest.mf es un fichero que contiene información sobre el componente, indica dónde se encuentran otros componentes y bibliotecas requeridas.

Los ficheros JAR pueden contener también archivos de recursos que no sean específicamente de código, por ejemplo, imágenes u otros formatos que sean necesarios durante la ejecución de las clases de los componentes.

La mayoría de entornos de desarrollo, como Eclipse, usan una utilidad llamada Ant para automatizar gran parte de las tareas requeridas a la hora de construir una aplicación, biblioteca o componente. La herramienta Ant se configura por medio de un archivo XML, el cual permite definir una serie de tareas que se pueden ejecutar de forma independiente y que Eclipse invoca durante la compilación, el empaque o la ejecución. Cada tarea se define como una secuencia de acciones simples que, al invocar la tarea, se ejecutan una tras otra. Al tratarse de un archivo XML, resulta muy sencillo hacer modificaciones para adaptar las acciones a las necesidades específicas que un determinado proyecto tenga. Generalmente, las tareas definidas por defecto son suficientes, pero a veces, si el componente dispone de elementos externos al JAR (archivos de configuración, documentación, etcétera), habrá que añadir alguna acción para copiar estos archivos en el directorio destino donde se generará todo el componente.

Para poder empaquetar un programa en Java, vamos a realizar una pequeña guía.

En primer lugar, tendremos que dirigirnos a nuestra aplicación en Eclipse y a pulsar con el botón derecho del ratón hasta encontrar la opción de *Export*.

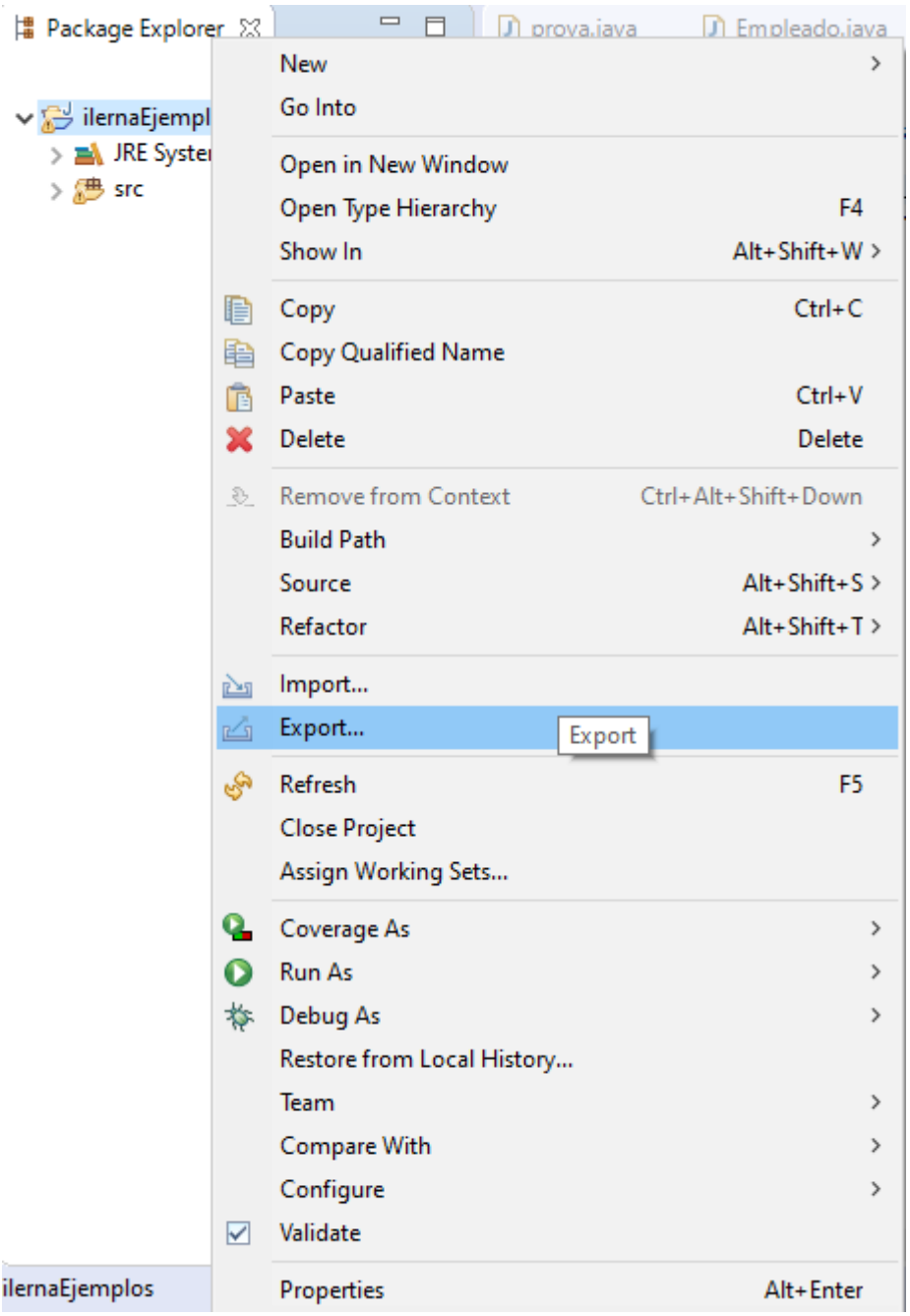


Ilustración 5. Captura de cómo exportar un proyecto.

Al pulsar *Exportar*, se nos tiene que abrir una ventana como esta:

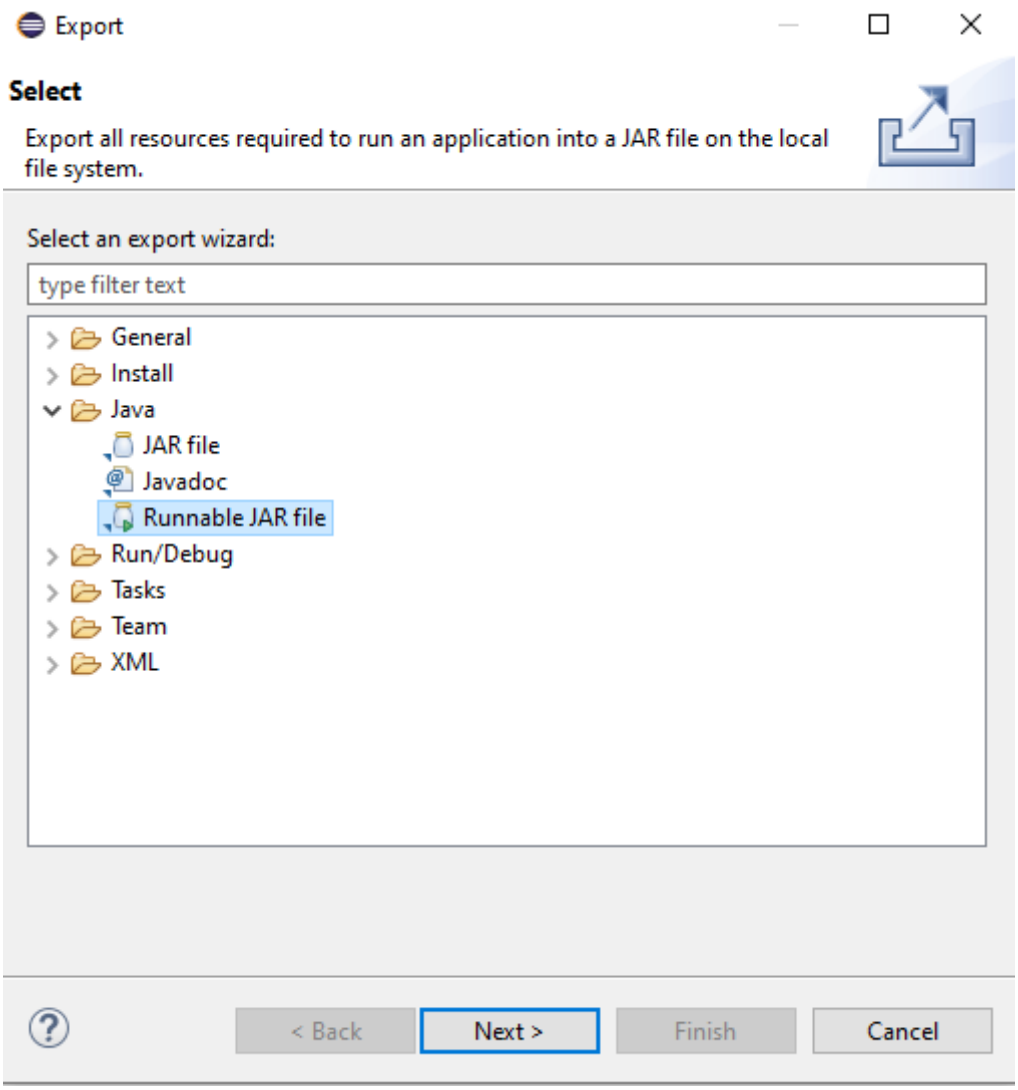


Ilustración 6. Proceso de exportación de un proyecto a un archivo JAR.

Seleccionaremos *Runnable Jar file* y nos aparecerá esto en la pantalla:

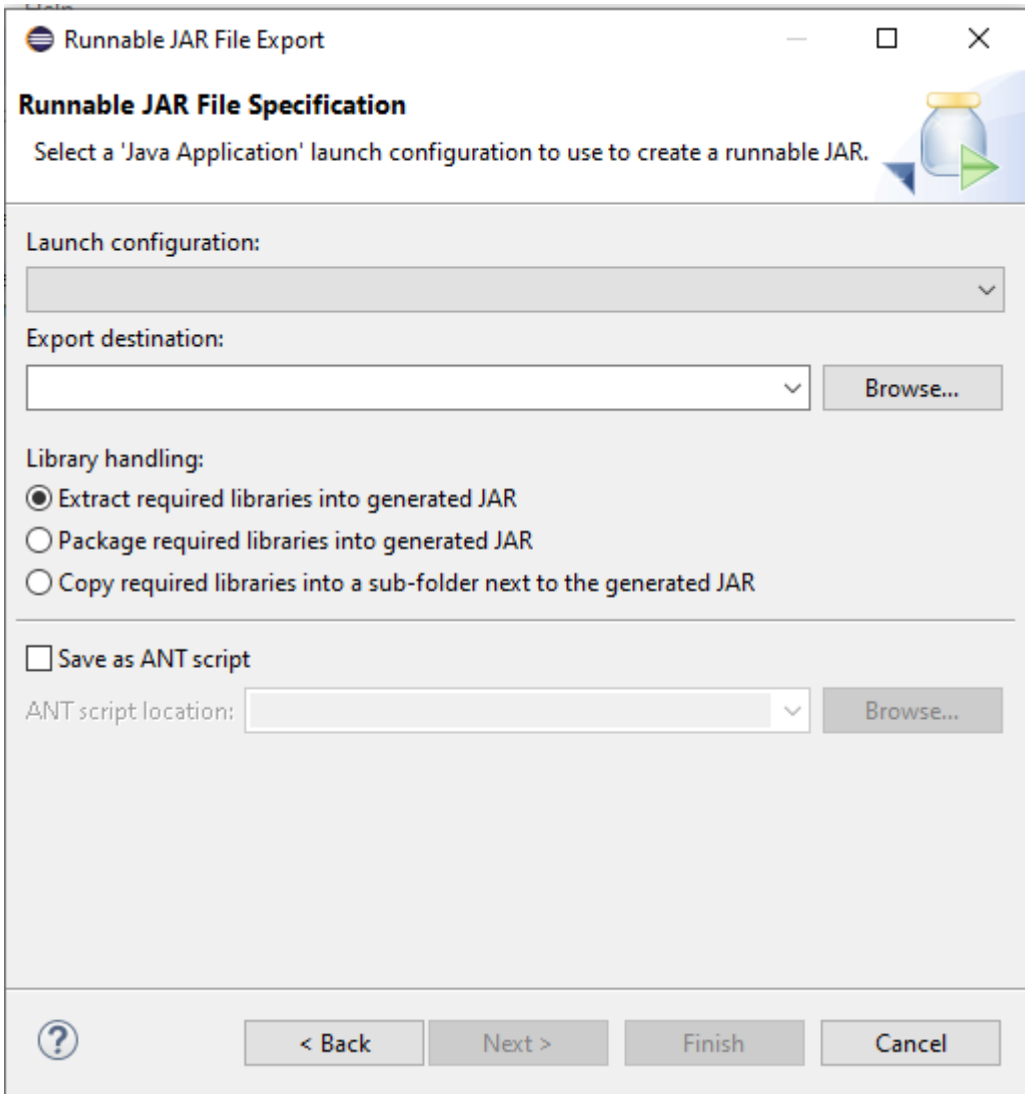


Ilustración 7. Captura de la ventana que permite elegir el proyecto a exportar y la ubicación.

En *Launch configuration* tendremos que seleccionar la primera opción del menú desplegable: *Configuración de lanzamiento*. A continuación, donde pone *Export destination* tendremos que indicar a Eclipse dónde queremos guardar el JAR en nuestro ordenador. Por último, en *Library handling* elegiremos *Extract required libraries into generated JAR:* esta opción transformará nuestra aplicación en un JAR.

Con estos pasos, tendríamos nuestra aplicación empaquetada. No se trata de pasos complicados, lo más complicado será realizar la tarea de desarrollo.

6.10 Prueba y documentación de componentes desarrollados

En este apartado, vamos a centrarnos en poner en práctica todo lo aprendido en esta lección. Vamos a crear un componente sencillo, a aprender a empaquetarlo y a usarlo como un componente en otra aplicación.

Siguiendo con el ejemplo del apartado 6.3, vamos a crear un contenedor que tendrá las funciones de una calculadora, la vista que el cliente usará para visualizar las funciones e interactuar con los métodos, y también un test para ver si los métodos implementados realmente están bien construidos. Será importante documentar todos los métodos y clases que vayamos creando, siguiendo la estructura de JavaDoc. Este tipo de aplicación es una aplicación de *bean stateless*, es de las estructuras más sencillas que nos encontraremos a la hora de crear un contenedor.

En primer lugar, nos dirigiremos a nuestro IDE, en este caso, Eclipse, y crearemos un proyecto nuevo de tipo EJB. Como hemos explicado en el apartado 6.3, tendremos que dirigirnos a *File > New > Other* y escribir EJB. Nos tendrá que salir una ventana que nos dará la opción de elegir el nombre del proyecto y le daremos a *Finalizar*.

Para poder desarrollar aplicaciones EJB, antes de empezar tendremos que importar la librería EJB a nuestro proyecto. La podemos descargar directamente desde este enlace:

https://repo1.maven.org/maen2/javax/ejb/ejb-api/3.0/ejb-api-3.0.jar

Después tenemos que estructurar el proyecto. Normalmente, se tienen que dividir por paquetes las clases para mantener el orden en los proyectos. En este caso, crearemos un paquete para la lógica de negocio que llamaremos "negocio" y otro donde implementaremos la lógica de negocio que llamaremos "bean". El primer paquete contendrá las clases que realizarán las acciones principales de nuestro componente; en cambio, el segundo contendrá las clases que representarán un objeto o entidad. La estructura tiene que ser algo parecido a esto:

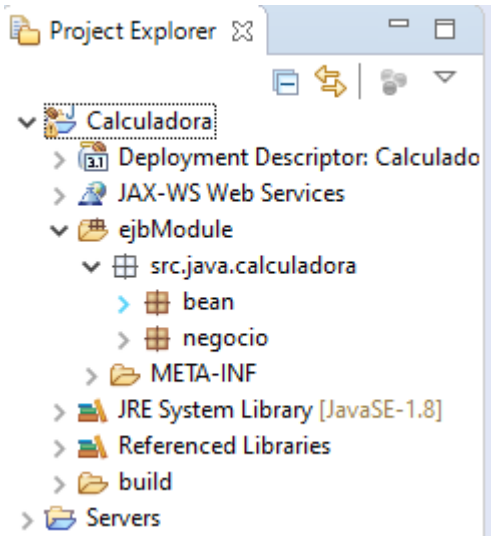


Ilustración 8. Estructura del proyecto por paquetes.

A continuación, crearemos la interfaz *CalculadoraNegocio*, que será la que contendrá la lógica de negocio de la calculadora, y *CalculadoraBean*, que implementará la lógica de negocio implementando *CalculadoraNegocio*. Las clases tendrían que estar así:

```java
package src.java.calculadora.negocio;

import javax.ejb.Remote;

/**
 * Interfaz que se encarga de la lógica de negocio de la calculadora
 * @author Laura
 *
 */
@Remote
public interface CalculadoraNegocio {
	/**
	 * Este método se encargará de realizar la suma entre dos valores
	 * recibidos por parámetro
	 * @param y
	 * @return devuelve un float
	 */
	float suma(float x, float y);

	/**
	 * Este método se encargará de realizar la resta entre dos valores que
	 * recibirá por parámetro
	 * @param x
	 * @param y
	 * @return devuelve un float
	 */
	float resta(float x, float y);

	/**
	 * Este método se encargará de realizar la multiplicación entre dos
	 * valores que recibirá por parámetro
	 * @param x
	 * @param y
	 * @return devuelve un float
	 */
	float multiplicacion(float x, float y);

	/**
	 * Este método se encargará de realizar la división entre dos valores
	 * que recibirá por parámetro
	 * @param x
	 * @param y
	 * @return
	 */
	float division(float x, float y);
}
```

```java
package src.java.calculadora.bean;
import javax.ejb.Stateless;
import src.java.calculadora.negocio.CalculadoraNegocio;
/**
 * Esta clase se encarga de la lógica de negocio de la calculadora implementa
 * la clase CalculadoraNegocio
 * @author Laura
 *
 */
@Stateless
public class CalculadoraBean implements CalculadoraNegocio {

	/**
	 * Este método se encargará de realizar la suma entre dos valores
	 * recibidos por parametro
	 */
	@Override
	public float suma(float x, float y) {
		return x + y;
	}

	/**
	 * Este método se encargará de realizar la resta entre dos valores que
	 * recibirá por parámetro
	 */
	@Override
	public float resta(float x, float y) {
		return x - y;
	}

	/**
	 * Este método se encargará de realizar la multiplicación entre dos
	 * valores que recibirá por parámetro
	 */
	@Override
	public float multiplicacion(float x, float y) {
		return x * y;
	}

	/**
	 * Este método se encargará de realizar la división entre dos valores
	 * que recibirá por parámetro
	 */
	@Override
	public float division(float x, float y) {
		return x / y;
	}

}
```

La implementación de cada método no tiene mucho misterio, cada método hará la operación matemática correspondiente: suma, resta, multiplicación y división, devolviendo un *float* como resultado.

Tras ello, tendremos que empaquetar nuestra aplicación: procederemos a pulsar encima del proyecto con el botón derecho del ratón.

Tendrá que aparecer una ventana y elegiremos la opción *Export*.

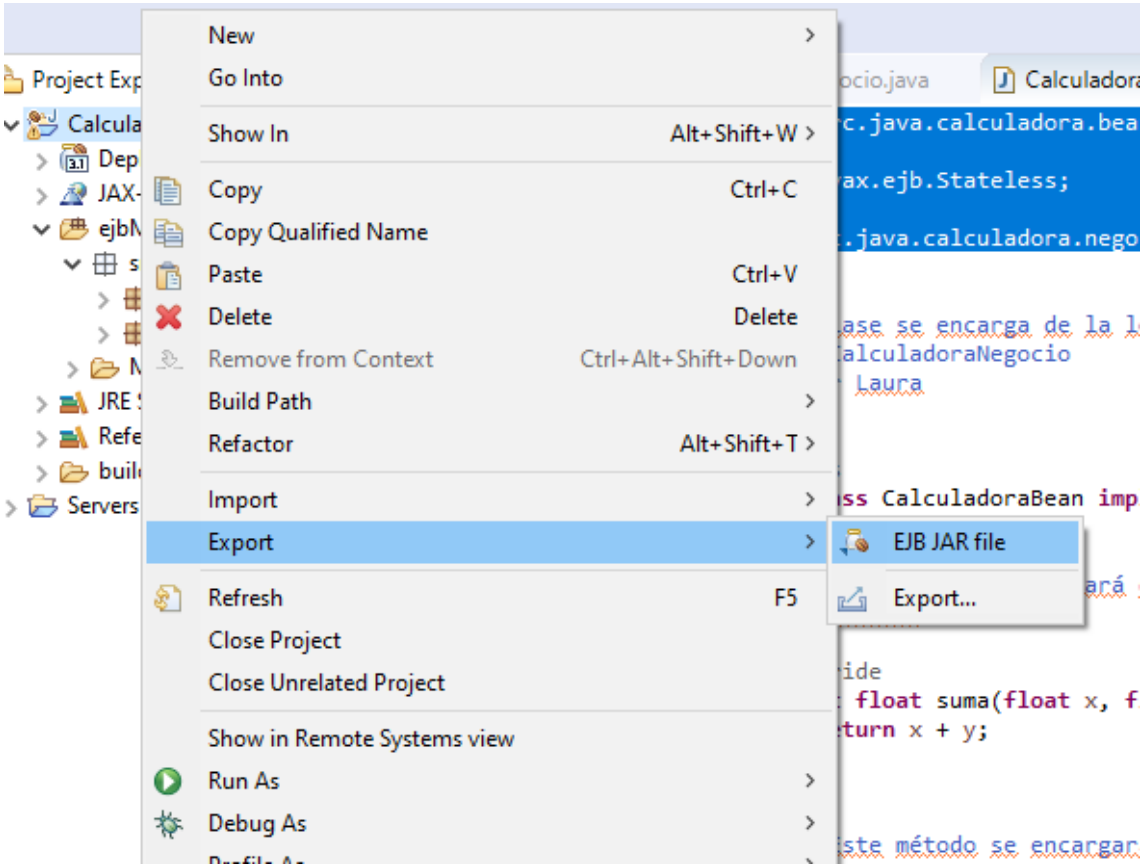


Ilustración 9. Captura de pasos de empaquetado de la aplicación.

Tal como indica la imagen, elegiremos la opción *EJB JAR file*. Seleccionaremos el proyecto que queremos exportar y le daremos un nombre. Debería ser algo parecido a esto:

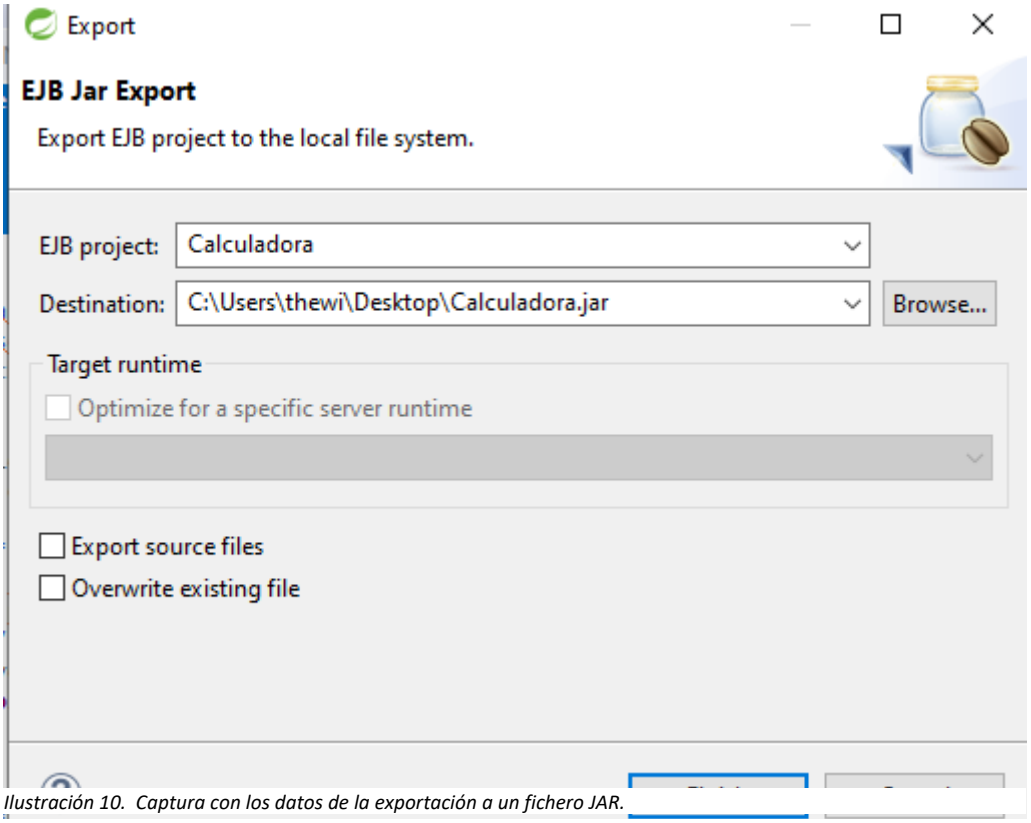


Ilustración 10. Captura con los datos de la exportación a un fichero JAR.

Ahora que tenemos el JAR, debemos crear una aplicación web que implementará nuestro componente. Para ello, iremos a *File > New > Dynamic Web Project:*

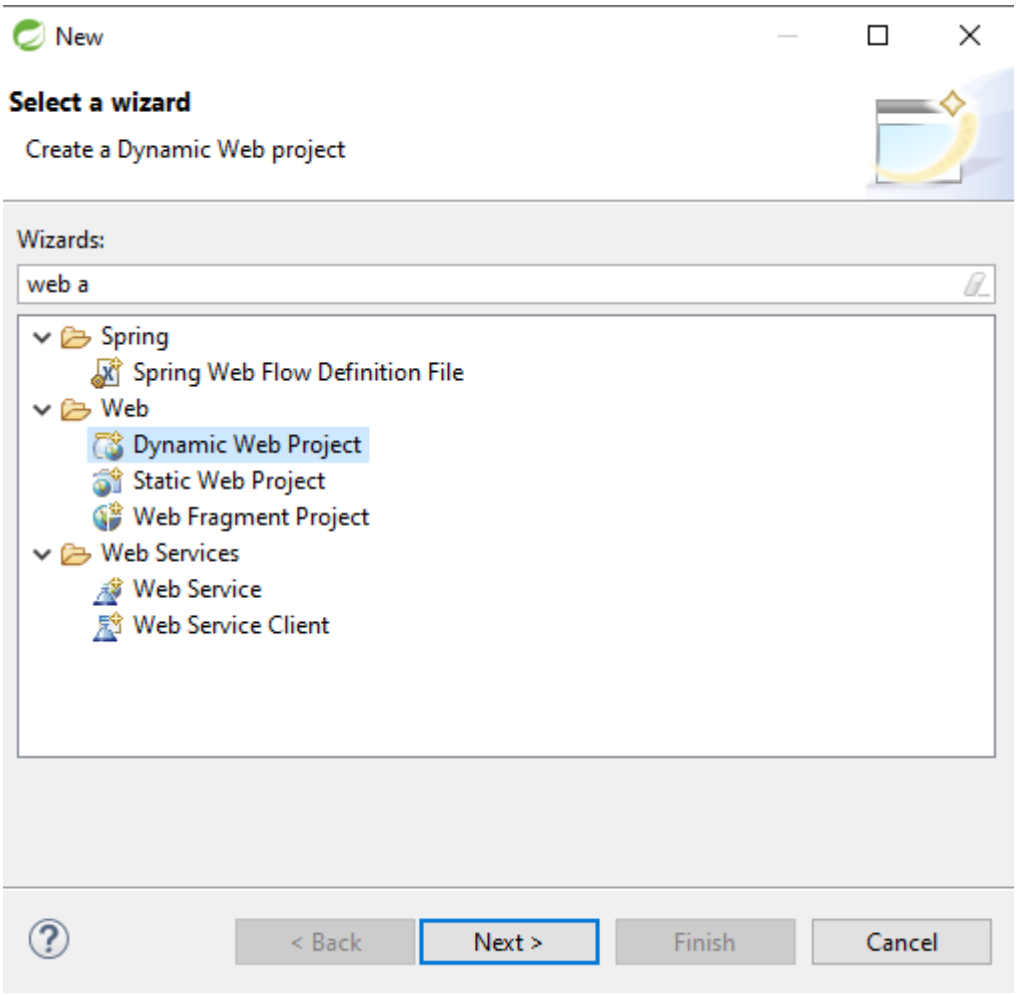


Ilustración 11. Creación de un proyecto dinámico Java.

Para poder continuar con la creación del proyecto, antes tenemos que importar la librería para poder gestionar *servlets*. La descargaremos de este enlace:

https://repo1.maven.org/maven2/javax/servlet/servlet-api/2.5/servlet-api-2.5.jar

A continuación, tendremos que crear un fichero JSP que será la página web donde podremos hacer las operaciones con la calculadora. Para ello, pulsaremos con el botón derecho en nuestro proyecto y seleccionaremos *New > JSP File*. Se tendrá que abrir una pantalla como esta:

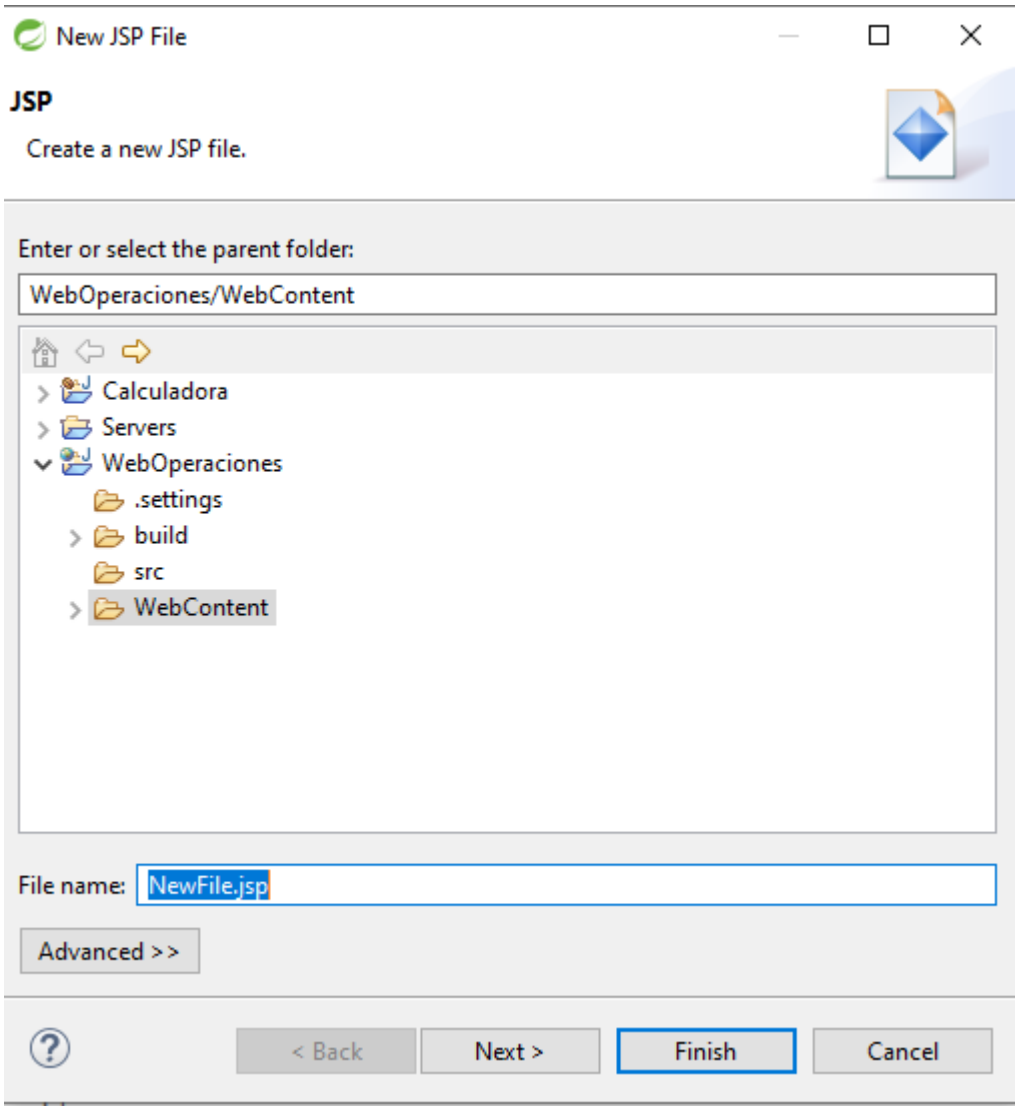


Ilustración 12. Creación de un fichero jsp.

Dentro del JSP, tendremos que añadir un formulario con las opciones de nuestra calculadora. Un JSP es un fichero que para construir aplicaciones web mediante Java que admite tanto HMTL como código Java embebido. En esta ocasión, solo utilizaremos el HTML. El código tiene que parecerse algo a esto.

```
<html>
    <head>
        <title>Calculadora</title>
    </head>
    <body>
        <h1>Operaciones básicas</h1>
        <hr>
        <form action="resultado.jsp" method="POST">
            <p>Introduzca el primer valor:
                <input type="text" name="num1" size="25"></p>
            <br>
            <p>Introduzca el Segundo valor:
                <input type="text" name="num2" size="25"></p>
            <br>

            <b>Seleccione una opción</b><br>
            <input type="radio" name="operacion" value ="suma">Suma<br>
            <input type="radio" name=" operacion" value
="resta">Resta<br>
            <input type="radio" name="operacion" value
="multi">Multiplicación<br>
            <input type="radio" name="operacion" value
="div">División<br>
            <p>
                <input type="submit" value="Enviar">
                <input type="reset" value="Borrar"></p>

        </form>
    </body>
```

El formulario estará ligado a una acción, es decir, los resultados de las operaciones realizadas en esta pantalla se redirigirán a una pantalla nueva llamada resultado.jsp, el cual tendremos que crear de la misma manera que hemos creado el anterior. Seguidamente, tendremos que importar el componente que hemos empaquetado anteriormente, y lo haremos de la misma manera que importaríamos una librería.

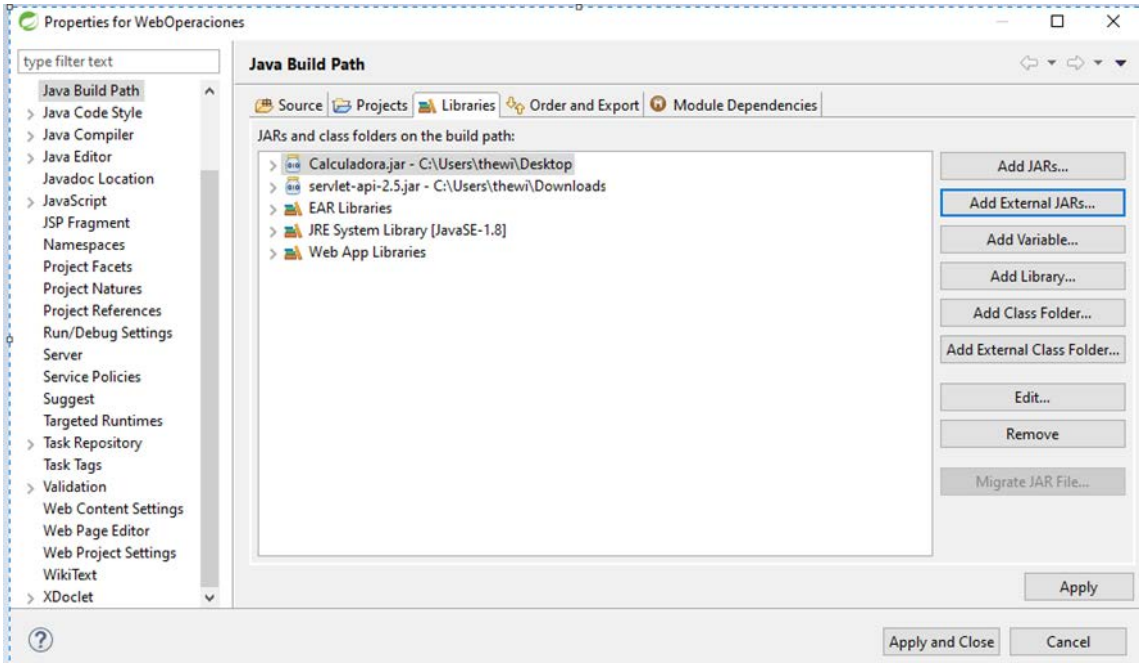


Ilustración 13. Captura de la importación del JAR calculadora.

El código que deberemos de implementar en resultado.jsp tendrá que ser algo parecido a esto:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ page import="src.java.calculadora.bean.*, javax.naming.*"%>

<%!
    private CalculadoraBean calculadora = null;
    float resultado = 0;

    public void jspInit() {
        try {

            InitialContext ic = new InitialContext();
            calculadora =
(CalculadoraBean)ic.lookup(CalculadoraBean.class.getName());
            System.out.println("Hemos cargado el bean de
CalculadoraBean");
        } catch (Exception ex) {
            System.out.println("Error:" + ex.getMessage());
        }
    }
    public void jspDestroy() {
       calculadora = null;
    }
    try {
        String s1 = request.getParameter("num1");
        String s2 = request.getParameter("num2");
        String s3 = request.getParameter("operacion");

        System.out.println(s3);

        if (s1 != null && s2 != null) {
            Float num1 = new Float(s1);
            Float num2 = new Float(s2);

            if (s3.equals("suma")) {
                resultado = calculadora.suma(num1.floatValue(),
num2.floatValue());
            } else if (s3.equals("resta")) {
                resultado = calculadora.resta(num1.floatValue(),
num2.floatValue());
            } else if (s3.equals("multi")) {
                resultado =
calculadora.multiplicacion(num1.floatValue(), num2.floatValue());
            } else {
                resultado = calculadora.division(num1.floatValue(),
num2.floatValue());
            }

%>
<p>
    <b>El resultado es:</b> <%= resultado %>
<p>
    <%
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    %>
```

1

2

3

Este fichero, como podemos apreciar, tiene embebido código Java. Los JSP se caracterizan por permitir añadir código Java en su fichero mediante los *tags* **<%** de apertura y **%>** de clausura. Dentro de estos *tags*, podemos añadir cualquier operación con Java.

En el punto 1, lo que hacemos es indicar el paquete de las clases que importaremos a continuación. Como queremos importar la clase *CalculadoraBean*, tendremos que añadir la ruta del paquete de esta clase en este *import*.

En el punto 2, tendremos que crear una nueva instancia de la clase *CalculadoraBean*. En el punto 3, declararemos un *InitialContext* que nos ayudará a cargar la instancia de *CalculadoraBean*. Esta pantalla se encargará de recoger los datos de la consulta: el valor "num1", el valor "num2" y el valor "operacion" para realizar la operación. En un *if*, comprobaremos el valor de operación para elegir a qué método tenemos que llamar. La operación la podremos hacer llamando a los métodos de la calculadora usando la instancia del *bean*. Con la incorporacion de la librería, lo que hemos hecho es añadir la funcionalidad sin necesidad de saber cómo está implementado el código.

Así tendremos creada nuestra aplicación, implementando un componente. Para poder ejecutar la aplicación, tenemos que hacer clic con el botón derecho del ratón encima de form.jsp y elegiremos *Run As > Other > Run on Server*, tal y como vemos en la imagen:

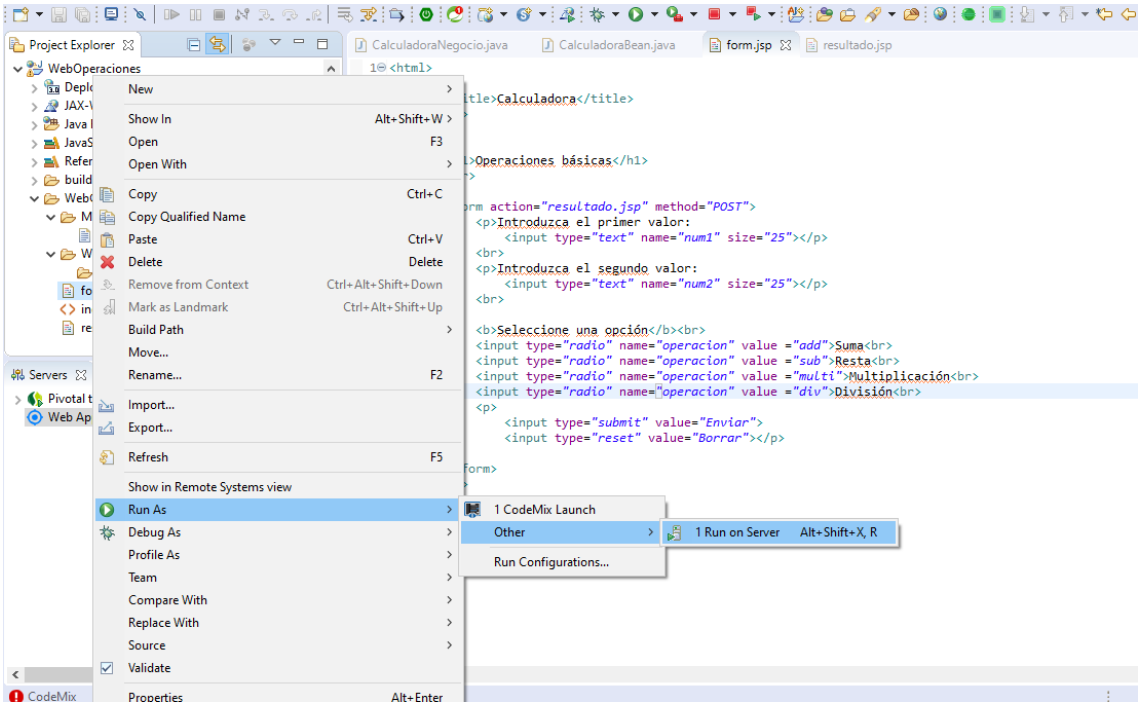


Ilustración 14. Captura para ejecutar la aplicación.

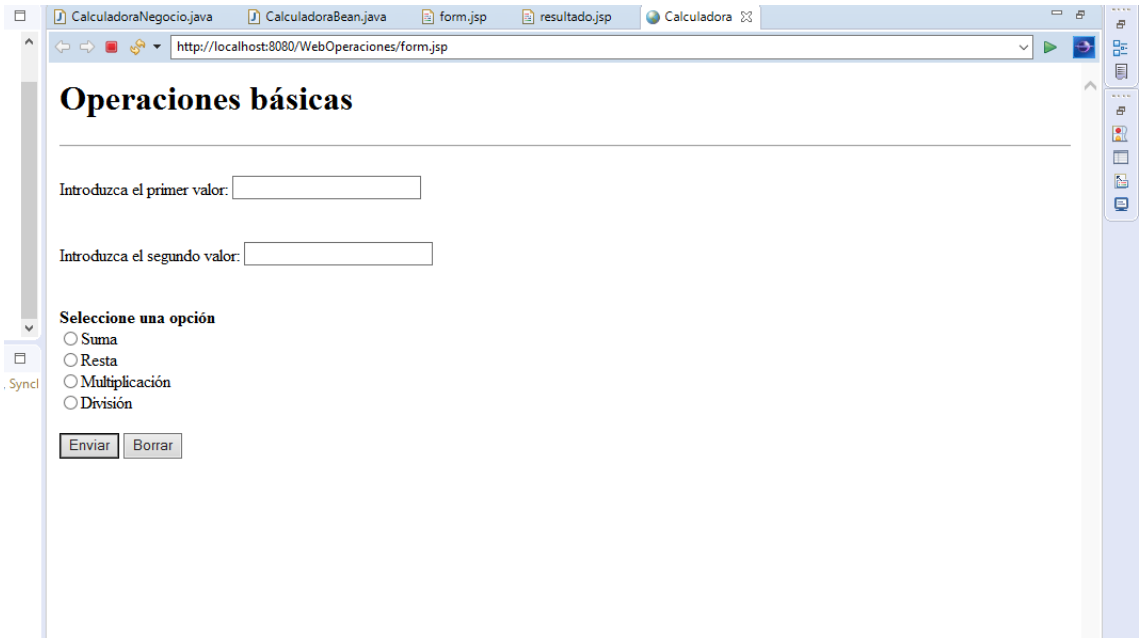


Ilustración 15. Captura de la aplicación funcionando.

Si todo va bien, veremos nuestra aplicación funcionando.

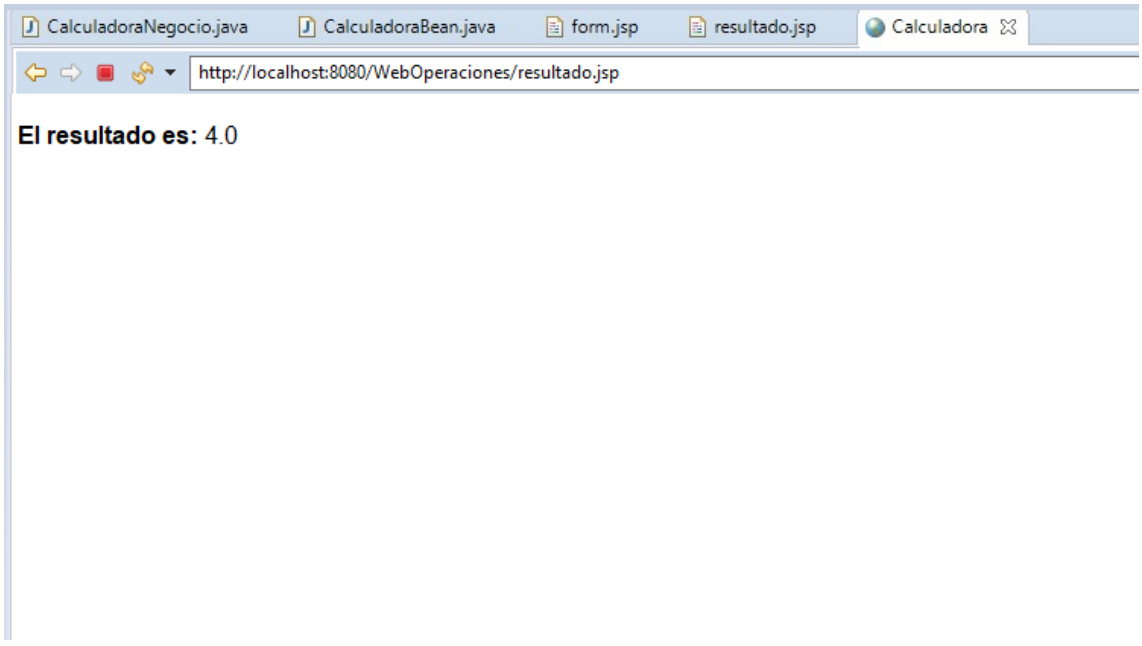


Ilustración 16. Captura con el resultado de la aplicación.

Bibliografía

Java Platform Standard Edition 8. "Documentación de la API Java NIO Files". Recuperado de: https://docs.oracle.com/javase/8/docs/api/java/nio/file/package-summary.html

Java Platform Standard Edition 7. "Documentación de la API File". Recuperado de: https://docs.oracle.com/javase/7/docs/api/java/io/File.html

ORACLE Java Documentation. "Parsing an XML file using SAX". Recuperado de: https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html

Tutorialspoint. "Java. Files and I/O". Recuperado de: https://www.tutorialspoint.com/java/java_files_io.htm

Baeldung. "Guide to BufferedReader". Recuperado de: https://www.baeldung.com/java-buffered-reader

Jenkov, J. *Java NIO Files*. Recuperado de: http://tutorials.jenkov.com/java-nio/files.html

Jenkov, J. *Java NIO vs. IO*. Recuperado de: http://tutorials.jenkov.com/java-nio/nio-vs-io.html

WC3. "Internationalization". Recuperado de: https://www.w3.org/International/questions/qa-what-is-encoding

Java Platform Standard Edition 7. "Documentación para el borrado de File". Recuperado de: https://docs.oracle.com/javase/tutorial/essential/io/delete.html

Java Platform Standard Edition 7. "Copying a file or directory". Recuperado de: https://docs.oracle.com/javase/tutorial/essential/io/copy.html

Java Platform Standard Edition 7. "Class BufferedWriter". Recuperado de: https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html

Java Platform Standard Edition 7. "Class FileOutputStream". Recuperado de: https://docs.oracle.com/javase/7/docs/api/java/io/FileOutputStream.html

Oracle Java Documentation. "File operations". Recuperado de: https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html

Baeldung. "Java. Rename or move a file". Recuperado de: https://www.baeldung.com/java-how-to-rename-or-move-a-file

Java Documentation. "Sax parsing". Recuperado de: https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html

Oracle Java Documentation. "Lesson: basic I/O". Recuperado de: https://docs.oracle.com/javase/tutorial/essential/io/

Eckel, B. *Piensa en Java*. Editorial Pearson.

JavaTpoint. "How to read XML file in Java". Recuperado de: https://www.javatpoint.com/how-to-read-xml-file-in-java

Flanagan, D. (2005). *Java in a nutshell*. O'Reilly.

Gupta, L. "JAXB read XML to Java object example". Recuperado de: https://howtodoinjava.com/jaxb/read-xml-to-java-object/

Vogel, L. *Java and XML. Tutorial*. Recuperado de: https://www.vogella.com/tutorials/JavaXML/article.html

Baeldung. "Parsing an XML file using SAX parser". Recuperado de: https://www.journaldev.com/1198/java-sax-parser-example

Baeldung. "XML libraries support in Java". Recuperado de: https://www.baeldung.com/java-xml-libraries

MySQL Documentation. Recuperado de: https://dev.mysql.com/doc/

Existdb Documentation. Recuperado de: https://exist-db.org/exist/apps/doc/documentation

JavaTpoint. "Java JDBC tutorial". Recuperado de: https://www.javatpoint.com/java-jdbc

MySQL Community Downloads. Recuperado de: http://dev.mysql.com/downloads/connector/j

Tutorialspoint. "JDBC: database connections". Recuperado de: https://www.tutorialspoint.com/jdbc/jdbc-db-connections.htm

Tutorialspoint. "JDBC: sample, example code". Recuperado de: https://www.tutorialspoint.com/jdbc/jdbc-sample-code.htm

Tutorialspoint. "JDBC tutorial". Recuperado de: https://www.tutorialspoint.com/jdbc/index.htm

JavaTpoint. "Transaction management in JDBC". Recuperado de: https://www.javatpoint.com/transaction-management-in-jdbc

Oracle Java Documentation. "Java JDBC API". Recuperado de: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

Hibernate ORM Documentation 5.4. Recuperado de: https://hibernate.org/orm/documentation/5.4/

Tutorialspoint. "Hibernate tutorial". Recuperado de: https://www.tutorialspoint.com/hibernate/index.htm

Journaldev. "Hibernate tutorial for beginners". Recuperado de: https://www.journaldev.com/2882/hibernate-tutorial-for-beginners#hibernate-mapping-xml-configuration

Baeldung. "Hibernate inheritance mapping". Recuperado de: https://www.baeldung.com/hibernate-inheritance

Baeldung. "Deleting objects with Hibernate". Recuperado de: https://www.baeldung.com/delete-with-hibernate

Janssen, T. *Hibernate tips: how to map an entity to multiple tables*. Recuperado de: https://thorben-janssen.com/hibernate-tips-how-to-map-an-entity-to-multiple-tables/

JournalDev. "Hibernate session merge, update, save, saveOrUpdate, persist example". Recuperado de: https://www.journaldev.com/3481/hibernate-session-merge-vs-update-save-saveorupdate-persist-example#hibernate-merge

Oracle Database Documentation. "Object-relational developer's guide". Recuperado de: https://docs.oracle.com/en/database/oracle/oracle-database/18/adobj/key-features-object-relational-model.html#GUID-8A38BA82-CCD5-4424-AE19-10A994E25B0E

Operational Database Management Systems. "Introduction to ODBMS". Recuperado de: http://www.odbms.org/introduction-to-odbms/definition/

Kyocera. "Conceptos sobre base de datos orientada a objetos". Recuperado de: https://www.kyoceradocumentsolutions.es/es/smarter-workspaces/business-challenges/paperless/conceptos-sobre-base-de-datos-orientada-a-objetos.html

Mkyong. "Connect to Oracle DB via JDBC driver". Recuperado de: https://mkyong.com/jdbc/connect-to-oracle-db-via-jdbc-driver-java/

Oracle Tutorial. "PL/SQL data types". Recuperado de: https://www.oracletutorial.com/plsql-tutorial/plsql-data-types/

Morgan, K.; Silberschatz, A.; Korth, H. y Sudarshan, S. (2002): *Fundamentos de bases de datos.* McGraw Hill.

Ha Minh, N. *Java connect to Oracle database via JDBC*. Recuperado de: https://www.codejava.net/java-se/jdbc/connect-to-oracle-database-via-jdbc

Database PL/SQL Language Reference. Recuperado de: https://docs.oracle.com/database/121/LNPLS/toc.htm

Oracle Database Documentation. "SQL object types and references". Recuperado de: https://docs.oracle.com/en/database/oracle/oracle-database/19/adobj/Sql-object-types-and-references.html#GUID-821233F6-ABA9-4100-B5E4-937F7DC57102

Oracle Tutorial. "PL/SQL anonymous block". Recuperado de: https://www.oracletutorial.com/plsql-tutorial/plsql-anonymous-block/

Database Migration Guide. "SQL translation using a JDBC driver". Recuperado de: https://docs.oracle.com/database/121/DRDAA/sqltran.htm#DRDAA29458

Oracle Database PL/SQL User's Guide and Reference. "Using PL/SQL with object types". Recuperado de: https://docs.oracle.com/cd/B19306_01/appdev.102/b14261/objects.htm#i11073

IBM Knowledge Center. "Lenguaje de consulta de objetos". Recuperado de: https://www.ibm.com/support/knowledgecenter/es/SSSHRK_3.9.0/com.ibm.networkmanagerip.doc_3.9/itnm/ip/wip/ref/concept/pen_ref_oql.html

IBM Knowledge Center. "Comparación entre XML y los modelos relacionales". Recuperado de: https://www.ibm.com/support/knowledgecenter/es/ssw_ibm_i_72/rzasp/rzaspxml3811.htm

Staken, K. *An introduction to the XML DB API*. Recuperado de: https://www.xml.com/pub/a/2002/01/09/xmldb_api.html

ECURED. "Bases de datos orientadas a objetos". Recuperado de: https://www.ecured.cu/Bases_de_datos_orientadas_a_objetos

GeeksForGeeks. "Difference between RDBMS and OODBMS". Recuperado de: https://www.geeksforgeeks.org/difference-between-rdbms-and-oodbms/

Gestión de Bases de Datos. Recuperado de: https://gestionbasesdatos.readthedocs.io/es/latest/Tema3/Teoria.html

Tutorialspoint. "PL/SQL tutorial". Recuperado de: https://www.tutorialspoint.com/plsql/index.htm

Tutorialspoint. "XPath tutorial". Recuperado de: https://www.tutorialspoint.com/xpath/index.htm

Tutorialspoint. "EJB tutorial". Recuperado de: https://www.tutorialspoint.com/ejb/index.htm

Harrington, J. (2000). *Object oriented database design clearly explained*.

Siegel, E.; Retter, A. *eXist: A no SQL document database and application platform.* O'Reilly.

Walmsley, P. *XQuery: search across a variety of XML data*. O'Reilly.

Rubinger, A. L. y Burke, B. *Enterprise JavaBeans 3.1*. O'Reilly.

Ray, E. T. *Learning XML*. O'Reilly.

St. Lauren, S. y Fitzgerald, M. (2005). *XML poquet reference*. O'Reilly.

Simpson, J. (2002). *XPath and XPointer*. O'Reilly.