TEMA 2: Manejo de conectores

Módulo Acceso a Datos

para los ciclos

Desarrollo de Aplicaciones Multiplataforma

2.1 Introducción

En este tema nos centraremos en el acceso desde ***Java*** a orígenes de datos relacionales. Para ello necesitamos usar los conectores. Los ***conectores*** son el software que se necesita para conectar con una base de datos relacional desde el código de un programa ***Java***.

Hay que tener en cuenta que actualmente la mayoría de los lenguajes de programación, entre ellos ***Java***, están orientados a objetos. Esto implica un esfuerzo extra para adaptar las estructuras relacionales a las orientadas a objetos y viceversa. A esas diferencias de estructuras se les llama desfase objeto-relacional y a la conversión necesaria se la conoce como mapeo.

El ***mapeo objeto-relacional*** se tratará más ampliamente en el siguiente tema.

Nota: A lo largo de todo este tema se hace referencia a ficheros y paquetes para descargar. Los nombres de muchos de ellos incluyen números que hacen referencia a su versión, por lo que lo más habitual será que los ficheros y paquetes que tú localices tengan una numeración distinta.

Incluimos en esta introducción unas indicaciones para trabajar con variables de entorno tanto en ***Linux*** como en ***Windows*** porque, aunque no está directamente relacionado con nuestros contenidos, necesitaremos usarlas durante la instalación de los diferentes ***SGBD***.

2.1.1 Variables de entorno en Linux

La declaración de variables de entorno en ***Linux*** (tanto desde la consola como en ficheros de configuración) se hace con la palabra ***export***:

```
export NOMBRE=valor
```

La referencia a una variable ya creada se hace anteponiendo el símbolo dólar (***$NOMBRE***). Así para mostrar el contenido de la variable anterior escribimos:

```
echo $NOMBRE
```

O para usarla como parte de otra asignación:

```
export VARIABLE2=$NOMBRE:valor2
```

Si establecemos los valores de las variables de entorno en el terminal, su valor estará vigente durante la sesión actual. Cuando queremos que esos valores se mantengan de forma persistente debemos declararlas como parte de alguno de los siguientes ficheros de texto:

- ***/etc/environment***
- ***/etc/profile***
- ***~/.bashrc***

Teniendo en cuenta que los dos primeros establecen los valores para todos los usuarios, mientras que el tercero sólo para un usuario.

Nota: En la ruta del último fichero aparece el símbolo de la virgulilla ***(~)*** que en los sistemas ***Linux*** representa la carpeta del usuario, por ejemplo, para el usuario ***administrador*** representa la carpeta ***/home/administrador*** y para el usuario ***antonio*** representa la carpeta ***/home/antonio***.

Cuando el valor de una variable de entorno de ***Linux*** haga referencia a varias rutas, como es el caso de ***PATH*** o ***CLASSPATH***, las separamos con el símbolo dos puntos ***(:)***

```
export PATH=/bin:/usr/bin
```

2.1.2 Variables de entorno en Windows

La declaración de variables de entorno en ***Windows*** se hace con la palabra ***set***:

```
set NOMBRE=valor
```

La referencia a una variable ya creada se hace encerrando su nombre entre símbolos tanto por ciento (***%NOMBRE%***). Así para mostrar el contenido de la variable anterior escribimos:

```
echo %NOMBRE%
```

O para usarla como parte de otra asignación:

```
set VARIABLE2=%NOMBRE%;valor2
```

Si establecemos los valores de las variables de entorno en el terminal, su valor estará vigente durante la sesión actual. Cuando queremos que esos valores se mantengan de forma persistente debemos modificarlo (dependiendo de la versión de ***Windows*** la localización puede variar ligeramente) en:

> ***Inicio / Panel de Control / Sistema / Configuración avanzada del sistema / Opciones avanzadas / Variables de entorno***

En esa ventana hay dos partes diferenciadas:

- En la parte superior trabajamos variables que afectan solamente a nuestro usuario.
- En la parte inferior trabajamos variables que afectan a todos los usuarios.

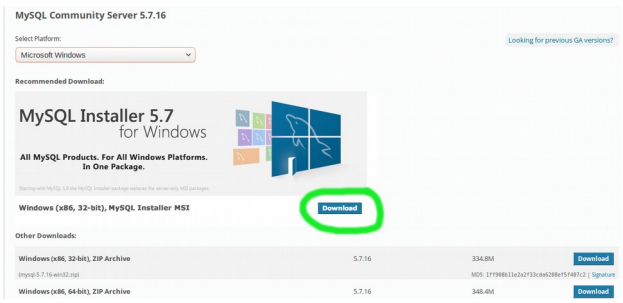
Cuando el valor de una variable de entorno de ***Windows*** haga referencia a varias rutas, como es el caso de ***PATH*** o ***CLASSPATH***, las separamos con el símbolo punto y coma ***(;)***

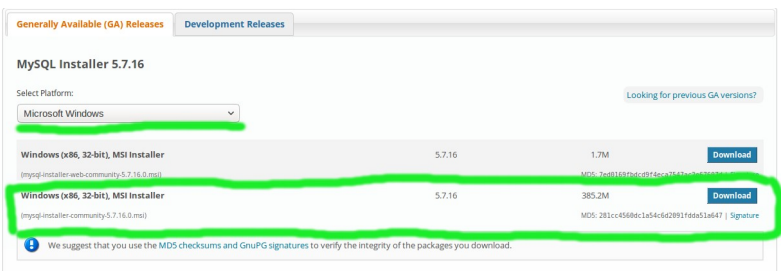
```
PATH=/bin;/usr/bin
```

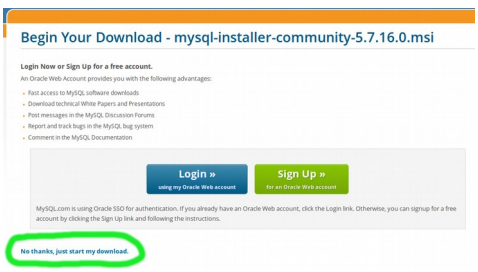
2.2 Instalación de Bases de datos

2.2.1 Instalación MySQL en Windows

La descarga la realizamos desde la web http://dev.mysql.com/downloads/ eligiendo el ***instalador .msi para Windows de 64 bits***. Para ello sigue la secuencia de imágenes:







Para instalar la aplicación sólo tienes que ejecutar el instalador ***.msi*** que te has descargado y seguir las siguientes instrucciones:

- Si parece que la instalación se paraliza probablemente haya una ventana escondida esperando que autorices la instalación.
- Acepta los términos.
- Elige la instalación personalizada (***Custom***), como en la siguiente imagen:
- Asegúrate de que sólo instalas el ***servidor de MySQL para 64 bits***. No necesitamos el resto de aplicaciones ni conectores.
- Continúa aceptando todas las opciones por defecto, salvo la contraseña del usuario ***root***, que debería ser ***root***.

Nota: La contraseña del usuario ***root*** de ***MySQL*** en un entorno de producción deberá ser una contraseña segura y conocida solamente por el administrador de la BD. Sin embargo, durante las clases vamos a usar todos la misma contraseña: ***root***.

Una vez terminada la instalación, desde un terminal de comandos, desplázate hasta la carpeta:

```
C:\Program Files\MySQL\MySQL Server 5.7\bin
```

Desde esa carpeta podrás ejecutar ***mysql***.

Nota: Para poder ejecutar ***mysql*** desde cualquier otro sitio tendrás que añadir la carpeta a la ***variable de entorno PATH***.

2.2.2 Instalación MySQL en Linux

Para instalar ***MySQL*** en una distribución de ***Linux*** bastará con instalar el paquete ***mysql-server***, que normalmente se encuentra en los repositorios de la propia distribución. Esto lo podemos hacer desde el ***Synaptic*** o bien, desde la línea de comandos:

```
sudo apt-get install mysql-server
```

La instalación inicial sólo se podrá ejecutar con el usuario ***root***, sin contraseña, y solamente desde el usuario ***root*** del sistema.

Será necesario crear un nuevo usuario de nombre y contraseña ***usuario***:

```
mysql> CREATE USER usuario IDENTIFIED BY 'usuario';
```

Nota: La contraseña de los usuarios de ***MySQL*** en un entorno de producción deberá ser una contraseña segura, sin embargo, durante este tema todos usaremos ***usuario***/***usuario***.

Las nuevas bases de datos se crean desde el usuario root y se le asignan privilegios al usuario. Por ejemplo:

```
mysql> CREATE DATABASE Liga;
mysql> GRANT ALL PRIVILEGES ON Liga.* TO usuario;
```

Para modificar una base de datos desde un fichero ***SQL***:

```
mysql -u usuario -p < liga.sql
```

o bien, desde dentro de ***mysql***:

```
mysql> source liga.sql
```

2.2.3 Descarga e instalación de Oracle en Ubuntu

Desde http://www.oracle.com/technetwork/es/indexes/downloads/index.html podemos bajar ***Oracle database***, concretamente la versión ***Oracle 11g express***, eligiendo el instalador que necesites (***Linux***, ***Win32*** y ***Win64***). La descarga proporciona un paquete ***ZIP*** que tendremos que desempaquetar. Dentro hallaremos un paquete ***RPM***. Los paquetes ***.rpm*** son paquetes instaladores para algunas distribuciones de ***Linux*** como ***Fedora***, ***CentOS***, ***OpenSuse*** o ***Mandriva***, mientras que ***Ubuntu*** usa paquetes ***.deb***, al igual que ***Debian*** que es de donde deriva ***Ubuntu***.

Conversión e instalación del paquete

Tendremos, por tanto, que convertir el paquete ***.rpm*** a ***.deb*** usando la utilidad de conversión ***alien***, que primero tendremos que instalar:

```
sudo apt-get install alien
sudo alien --scripts paquete.rpm
```

Nota: La conversión puede tardar unos minutos, así que conviene tener paciencia. Con este tema se proporciona el fichero ***.deb***, por tanto no es necesaria la conversión.

Una vez que tenemos el paquete ***.deb***, procedemos a instalar con:

```
sudo dpkg -i paquete.deb
```

Configuración

Presta mucha atención a los mensajes que genera la instalación. Es probable que aparezca, entre ellos, que no se ha podido terminar la configuración automática:

```
Executing post-install steps...

/var/lib/dpkg/info/oracle-xe.postinst: línea 114: /sbin/chkconfig: No existe el archivo o el directorio

You must run '/etc/init.d/oracle-xe configure' as the root user to configure the database.

Procesando disparadores para libc-bin (2.24-3ubuntu1) ...
```

En ese caso y siguiendo las instrucciones, ejecutamos:

```
sudo /etc/init.d/oracle-xe configure
```

El resultado tendrá un aspecto como el que sigue, donde se hacen varias preguntas. Cuando queramos aceptar la respuesta ofrecida entre corchetes pulsamos Enter, si no, escribimos la respuesta. Puedes aceptar todas las respuestas por defecto, pero es importante ***aportar una contraseña*** que tendremos que recordar (se recomienda usar ***root***, aunque en un sistema en producción habrá que usar una contraseña robusta).

```
administrador@VirtualBoxUbuntu:/sbin$ sudo /etc/init.d/oracle-xe configure

Oracle Database 11g Express Edition Configuration
-------------------------------------------------
This will configure on-boot properties of Oracle Database 11g Express
Edition.  The following questions will determine whether the database should
be starting upon system boot, the ports it will use, and the passwords that
will be used for database accounts.  Press <Enter> to accept the defaults.
Ctrl-C will abort.

Specify the HTTP port that will be used for Oracle Application Express [8080]:

Specify a port that will be used for the database listener [1521]:

/etc/init.d/oracle-xe: line 405: /bin/awk: No such file or directory
Specify a password to be used for database accounts.  Note that the same
password will be used for SYS and SYSTEM.  Oracle recommends the use of
different passwords for each database account.  This can be done after
initial configuration:
Confirm the password:

Do you want Oracle Database 11g Express Edition to be started on boot (y/n) [y]:

Starting Oracle Net Listener...Done
Configuring database...Done
Starting Oracle Database 11g Express Edition instance...Done
Installation completed successfully.
administrador@VirtualBoxUbuntu:/sbin$
```

Nota: Para volver a configurar, editamos o eliminamos el fichero ***/etc/default/oracle-xe***

Ejecución de Oracle

La aplicación aparece en el buscador por "***run sql command line***". El ejecutable (***sqlplus***) se encuentra en la siguiente ruta:

```
/u01/app/oracle/product/11.2.0/xe/bin
```

Ejecución de Oracle desde el terminal

Para una cómodo ejecución desde el terminal tendremos que hacer los siguientes ajustes:

- Añadir la ruta de ***sqlplus*** en la variable de entorno ***PATH***. Tendrás que editar el fichero ***/etc/environment*** e incluir la ruta indicada en el párrafo anterior.
- Añadir las variables ***ORACLE_HOME*** y ***ORACLE_SID***, en el fichero ***/etc/environment*** con los siguientes valores:

```
ORACLE_HOME="/u01/app/oracle/product/11.2.0/xe"
ORACLE_SID="XE"
NLS_LANG="SPANISH_SPAIN.AL32UTF8"
```

  Hay que reiniciar la sesión para que sea leído el fichero ***/etc/environment***.

- Finalmente, para que nuestro ***usuario Linux*** tenga privilegios para la ejecución de ***Oracle*** con todos los privilegios necesitamos añadirlo al ***grupo Linux dba***, con:

```
sudo adduser usuario dba
```

Iniciar Oracle

Una vez añadidas estas rutas, podemos llamar a la consola de ***Oracle*** con:

```
sqlplus
```

Post-instalación

La consola de Oracle en ***Ubuntu***, inicialmente, no dispone de historial de comandos ni permite su reedición. Esto se puede solucionar instalando desde los repositorios el paquete ***rlwrap***, tras lo cual debemos iniciar la consola con la orden:

```
rlwrap sqlplus
```

Nota: Se puede crear un alias para esta orden (***alias oracle='rlwrap sqlplus'***) en ***~/.bashrc***.

Si, al trabajar dentro de la consola de ***Oracle***, recibes el mensaje:

```
error ORA-01034: ORACLE not available
```

debes ejecutar (desde la consola de ***Oracle***) el comando:

```
SQL> startup
```

Usuario y contraseña

Si has seguido las instrucciones debes usar el usuario: ***root as sysdba*** y la contraseña: ***root***.

```
oracle root/root as sysdba
```

Para cada BD creamos un usuario con nombre y contraseña iguales a la BD y con privilegios.

```
SQL> CREATE USER liga IDENTIFIED BY liga;
SQL> GRANT ALL PRIVILEGES TO liga;
```

Dentro de un fichero ***SQL*** empezamos conectándonos con el usuario adecuado:

```
connect liga/liga
```

y lo ejecutamos desde la línea de comandos del sistema operativo:

```
oracle root/root as sysdba < liga.oracle.sql
```

o desde la línea de comandos de ***Oracle***:

```
SQL> @liga.oracle.sql
```

2.3 Bases de Datos relacionales embebidas

Los grandes gestores de ***BBDD*** son ***Oracle*** y ***MySQL***. Sin embargo, para pequeñas aplicaciones que no manejan grandes cantidades de datos podemos usar ***BBDD*** embebidas en las que el motor de la ***BD*** está incrustado en la aplicación y es exclusivo para ella. Algunas de ellas son:

- ***SQLite***
- ***Apache Derby***
- ***HSQLDB***
- ***H2***

2.3.1 SQLite

Es un ***SGBD*** ligero y de dominio público, escrito en ***C***. Guarda la base de datos en ficheros de modo que es fácil trasladarla con la aplicación. Tiene modo consola e implementa la mayor parte del estándar ***SQL-92***.

- Web de descarga: http://www.sqlite.org/download.html
- Instalación:
  - En ***Windows***, descargar y descomprimir: ***sqlite-tools-win32-x86-3250300.zip***
  - En ***Linux***:

```
sudo apt-get install sqlite3
```

- Ejecución (abre una ***BD***, si no existe, la crea):

```
sqlite3 ruta\nombreBD
```

- Crear una ***BD*** desde un fichero que tiene el código ***SQL***:

```
sqlite3 nombreBD < fichero.sql
```

- Podremos usar los comandos ***SQL*** en el prompt de ***SQLite3***
- Para salir:

```
...> .quit
```

2.3.2 Apache Derby

Es un ***SGBD*** de tamaño reducido, escrito en ***Java***. Soporta ***SQL*** e integra ***JDBC***. Soporta también el paradigma cliente-servidor.

- Web de descarga: http://db.apache.org/derby/derby_downloads.html

INSTALACIÓN EN WINDOWS

- En ***Windows***, descargar y descomprimir: ***db-derby-10.14.2.0-bin.zip***
- Copiar la carpeta (generada al descomprimir) en el sitio adecuado, por ejemplo:

```
D:\Archivos de Programa
```

- Añadir a la variable ***CLASSPATH*** las siguientes rutas:

```
carpeta de instalación\lib\derby.jar
carpeta de instalación\lib\derbytools.jar
```

Por ejemplo, si la copiaste en la ruta indicada como ejemplo será:

```
D:\Archivos de Programa\db-derby-10.14.2.0-bin\lib\derby.jar
D:\Archivos de Programa\db-derby-10.14.2.0-bin\lib\derbytools.jar
```

- Añadir a la variable ***PATH*** la siguiente ruta:

```
carpeta de instalación\bin
```

Por ejemplo, si la copiaste en la ruta indicada como ejemplo será:

```
D:\Archivos de Programa\db-derby-10.14.2.0-bin\bin
```

INSTALACIÓN EN LINUX

```
sudo apt-get install derby-tools
```

CONEXIÓN A BBDD

Para conectarte al gestor de BBDD ***Apache Derby*** basta con ejecutar con la orden: ***ij***

y aparecerá el prompt de ***Apache Derby***, donde podemos usar las siguientes instrucciones:

- Para crear una ***BD***:

```
ij> connect 'jdbc:derby:MICARPETA\MIBASE;create=true';
```

- Para abrir una ***BD*** ya creada:

```
ij> connect 'jdbc:derby:MICARPETA\MIBASE';
```

- Para salir:

```
ij> exit;
```

Si tenemos el código ***SQL*** en un fichero de texto podemos generar la BD directamente:

- Crear una ***BD*** desde un fichero que tiene el código ***SQL***:

```
ij < fichero.sql
```

2.3.3 HSQLDB

Es el ***SGBD*** que usa ***OpenOffice***, escrito en ***Java*** y compatible con ***ANSI-92*** y ***SQL2008***.

- Web de descarga: http://sourceforge.net/projects/hsqldb/files/
- Fichero a descargar y descomprimir: ***hsqldb-2.4.1.zip***

Al descomprimir creará una subcarpeta ***\hsqldb-2.4.1\hsqldb***. Podemos anular la carpeta \ ***hsqldb-2.4.1*** para aligerar la estructura, en cualquier caso la carpeta de instalación será ***....hsqldb***

INSTALACIÓN EN WINDOWS

- Crear una carpeta para las BBDD en ***CarpetaInstalación\data***
- Ejecutar: ***runUtil DatabaseManager***

Nota: Añade su carpeta (***CarpetaInstalación\bin***) a la variable de entorno ***PATH***.

INSTALACIÓN EN UBUNTU

- Mover la carpeta ***hsqldb*** a ***/opt/hsqldb***
- Añadir a la variable ***CLASSPATH*** la ruta: ***/opt/hsqldb/lib/hsqldb.jar***

```
export CLASSPATH=$CLASSPATH:/opt/hsqldb/lib/hsqldb.jar
```

Nota: Para que este cambio sea permanente añadimos en el fichero ***/etc/environment*** la línea:

CLASSPATH="/opt/hsqldb/lib/hsqldb.jar"

- Crear la carpeta para las Bases de Datos en (dentro de ella se creará una carpeta para cada base de datos):

```
mkdir ~/db/Hsqldb
```

- Ejecutar desde un terminal:

```
java org.hsqldb.util.DatabaseManager
```

Nota: Para no escribir ***java org.hsqldb.util.DatabaseManager*** en sucesivas ejecuciones de esta aplicación, podemos copiar el fichero ***hsqldb.sh*** que se proporciona (se trata de un fichero de texto con exclusivamente una línea con la orden anterior) en una carpeta que esté incluida en la variable ***PATH***, por ejemplo ***/usr/local/bin***, y ya lo podremos ejecutar con solo:

```
hsqldb.sh
```

Nota: Para poder colocar un fichero en la carpeta ***/usr/local/bin*** tendremos que hacerlo como ***root***, pero una vez copiado será necesario que cambiemos el propietario del fichero a nuestro usuario, de lo contrario no nos dejará ejecutarlo: ***chown usuario:usuario hsqldb.sh***

CREACIÓN DE BASES DE DATOS

Una vez ejecutado (en ***Linux*** y ***Windows***) aparece el entorno gráfico. En la ventana que sale:

- ***Setting Name***: el nombre de la conexión.
- ***Type***: Seleccionamos ***HSQL Database Engine Standalone***.
- ***Driver***: Lo dejamos como está: ***org.hsqldb.jdbcDriver***.
- ***URL***: Carpeta de la BD: ***jdbc:hsqldb:file:/home/usuario/db/Hsqldb/MiBase/*** (Muy importante la barra final)

Nota: El acceso a una BD ya creada se hace de la misma forma.

2.3.4 H2

Es un ***SGBD*** relacional programado en ***Java***.

- Web de descarga: http://www.h2database.com/html/download.html
- Fichero a descargar y descomprimir: ***h2-2017-06-10.zip***

Al descomprimir creará una carpeta ***\h2***

INSTALACIÓN EN WINDOWS

- Sólo hay que ubicar la carpeta en el lugar deseado y añadir la ruta ***….\h2\bin*** a la variable de entorno ***PATH***.
- Se ejecuta el fichero ***h2.bat***

Instalación en Ubuntu

- Mover la carpeta ***h2*** a ***/opt/h2***
- Dar permiso de ejecución al fichero ***/opt/h2/bin/h2.sh*** con el comando

```
sudo chmod +x /opt/h2/bin/h2.sh
```

- Añadir a la variable de entorno ***PATH***, la ruta ***/opt/h2/bin***
- Se ejecuta el fichero ***h2.sh***

Creación de Bases de Datos

Al ejecutarlo se abre el navegador web con la consola de administración de ***H2***.

En el campo controlador escribimos:

```
org.h2.jdbcDriver
```

En el campo ***URL JDBC*** escribimos:

```
jdbc:h2:rutaAbsoluta/nombreBD
```

Si la BD no existe, la crea. A partir de aquí podremos trabajar con la BD de modo gráfico.

2.4 Protocolos de acceso a BBDD

Existen dos normas de conexión a BBDD ***SQL***:

- ***ODBC***: (Open Database Conectivity) Es un estándar de acceso a BBDD desarrollado por ***Microsoft*** que incluye una ***API***. Cada sistema de base de datos compatible tiene que proporcionar una biblioteca para enlazar. ***ODBC*** tiene problemas de seguridad y robustez cuando se usa desde aplicaciones ***Java***, porque su interfaz está escrita en ***C***. Además es duro de aprender.
- ***JDBC***: (Java Database Conectivity) Es una librería estándar que proporciona una arquitectura y una inferfaz distinta para cada base de datos (***driver***). En el modelo de dos capas una aplicación ***Java*** pide (a través del ***driver***) los datos directamente al ***SGBD***, que puede residir en la misma o en otra máquina. En el modelo de tres capas una ***applet*** o navegador se comunica con el servidor de aplicaciones que es el que tiene el ***driver***. Existen 4 tipos de drivers:
  - ***JDBC-ODBC Bridge***: permite el acceso a BBDD ***JDBC*** mediante un driver ***ODBC***. Requiere instalar ***ODBC*** en el cliente.
  - ***Native***: Traduce las llamadas de ***JDBC*** en llamadas propias del motor de la BD. Requiere instalar código del cliente de la BD y del SO.
  - ***Network***: Controlador de Java que usa protocolo de red (como ***HTTP***) para comunicar con un servidor de BD. Traduce las llamadas ***JDBC*** en llamadas del protocolo de red. No requiere instalación en el cliente.
  - ***Thin***: Traduce las llamadas ***JDBC*** en llamadas propias del protocolo de red que usa el motor de la BD. No requiere instalación en el cliente.

    Lo normal es usar drivers tipo ***Network*** o tipo ***Thin***, los otros se usan cuando no hay más remedio.

2.4.1 JDBC

Cuando usamos un conector ***JDBC*** las clases más importantes son:

- ***Driver***: Es el que permite conectarse a un ***SGBD***. Cada gestor tiene su propio ***driver*** en forma de librería ***Java***. Será necesario descargarlo, situarlo en un sitio conocido y comunicar ese sitio. Si ejecutamos desde Eclipse lo anotaremos desde la ***vía de construcción***, si lo ejecutamos en línea lo anotaremos en la variable de entorno ***CLASSPATH***.
- ***DriverManager***: Es el que gestiona todos los ***drivers*** instalados en el sistema.
- ***Connection***: Es la representación de la conexión a la BD.
- ***Statement***: Sentencia ***SQL*** sin parámetros.
- ***PreparedStatement***: Sentencia ***SQL*** con parámetros.
- ***CallableStatement***: Sentencia ***SQL*** con parámetros de entrada y salida.
- ***ResultSet***: Contiene la respuesta a una sentencia ***SQL***.

El funcionamiento de un programa con ***JDBC*** sigue los siguientes pasos:

- Cargar el driver ***JDBC*** que corresponda. Por ejemplo, el driver ***JDBC*** para ***MySQL***:

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

- Crear ***Connection***. Por ejemplo:

```
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/NombreBD", "usuario", "psswd");
```

- Crear ***Statement***.

```
Statement sentencia = conexion.createStatement();
```

- Ejecutar la consulta con ***Statement***.

```
ResultSet respuesta = sentencia.executeQuery("SELECT * FROM Jugadores");
```

- Recuperar los datos del ***ResultSet***.

```
respuesta.getString("nombre");
```

- Liberar los objetos ***ResultSet***, ***Statement*** y ***Connection***.

```
respuesta.close();
sentencia.close();
conexion.close();
```

CONECTAR MYSQL CON JDBC

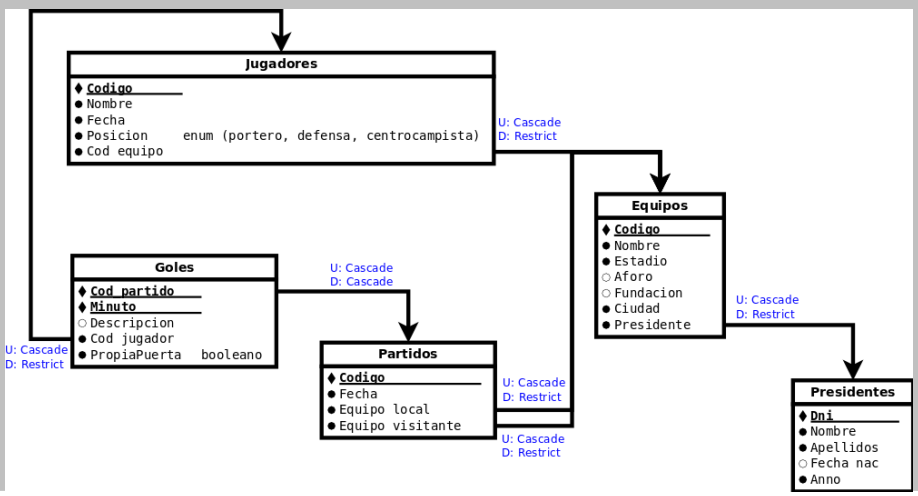
Para que esto funcione necesitamos el ***driver***, que se encuentra en una librería de ***MySQL*** para ***Java***. Tenemos que descargarla, situarla en un sitio conocido e informar a ***Eclipse*** (o al compilador de ***Java*** si lo hacemos en línea) del lugar donde se encuentra esa librería.

Desde la dirección http://www.mysql.com/products/connector/ podemos descargar el ***driver JDBC*** para ***MySQL***. Descargamos un fichero ***.deb*** que NO DEBEMOS EJECUTAR, solamente desempaquetar para buscar el fichero ***mysql-connector-java-8.0.13.jar***, que es el que nos interesa.

Hay que situar la librería en un lugar conocido (se recomienda ***/lib/mysql***) e informar a ***Eclipse*** de que se encuentra ahí:

```
CLICK-DCHO SOBRE EL PROYECTO/PROPIEDADES/VÍA DE CONSTRUCCIÓN JAVA/BIBLIOTECA DEL SISTEMA/AÑADIR JARS EXTERNOS
```

Nota: Véase el ejemplo ***AD01_JDBCMySQL.java*** que ejecuta una consulta sobre ***MySQL***. Este ejemplo trabaja sobre la BD Liga, cuyo esquema relacional es el siguiente:



Conectar a Oracle con JDBC

Nota: Véase el ejemplo ***AD06_JDBCOracle.java*** que ejecuta una consulta sobre ***Oracle***.

Conectar a SQLITE con JDBC

Desde la dirección https://bitbucket.org/xerial/sqlite-jdbc/downloads podemos descargar el ***driver JDBC*** para ***SQLite***: el fichero ***sqlite-jdbc-3.23.1.jar***.

Hay que situar la librería en un lugar conocido (se recomienda ***/lib/sqlite***) e informar a ***Eclipse*** de que se encuentra ahí:

```
CLICK-DCHO SOBRE EL PROYECTO/PROPIEDADES/VÍA DE CONSTRUCCIÓN JAVA/BIBLIOTECA DEL SISTEMA/AÑADIR JARS EXTERNOS
```

Ahora el ***driver*** y el ***conector*** serán:

```
Class.forName("org.sqlite.JDBC");
Connection conexion = DriverManager.getConnection("jdbc:sqlite:ruta/nombreBD");
```

Nota: Véase el ejemplo ***AD02_JDBCSQLite.java*** que ejecuta una consulta sobre ***SQLite***.

Conectar a Apache Derby con JDBC

En el fichero ***.zip*** que hemos descargado para la instalación de ***Apache derby*** en ***Windows*** se encuentra el ***driver JDBC*** para ***Apache derby***: el fichero ***derby.jar***.

Hay que situar la librería en un lugar conocido (se recomienda ***/lib/derby***) e informar a ***Eclipse*** de que se encuentra ahí:

```
CLICK-DCHO SOBRE EL PROYECTO/PROPIEDADES/VÍA DE CONSTRUCCIÓN JAVA/BIBLIOTECA DEL SISTEMA/AÑADIR JARS EXTERNOS
```

Ahora el ***driver*** y el ***conector*** serán:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Connection conexion = DriverManager.getConnection("jdbc:derby:ruta/nombreBD");
```

Nota: Véase el ejemplo ***AD03_JDBCDerby.java*** que hace una consulta sobre ***Apache derby***.

Conectar a HSQLDB con JDBC

En el fichero ***.zip*** que hemos descargado para la instalación de ***Hsqldb*** se encuentra el ***driver JDBC*** para ***Hsqldb***: el fichero ***hsqldb.jar***.

Hay que situar la librería en un lugar conocido (se recomienda ***/lib/hsqldb***) e informar a ***Eclipse*** de que se encuentra ahí:

```
CLICK-DCHO SOBRE EL PROYECTO/PROPIEDADES/VÍA DE CONSTRUCCIÓN JAVA/BIBLIOTECA DEL SISTEMA/AÑADIR JARS EXTERNOS
```

Ahora el ***driver*** y el ***conector*** serán:

```
Class.forName("org.hsqldb.jdbcDriver");
Connection conexion =
DriverManager.getConnection("jdbc:hsqldb:RUTA_ABSOLUTA/nombreBD/");
```

Nota: Observa la barra final tras el nombre de la Base de datos. Si la pones se creará una carpeta con la BD. Si no la pones, los ficheros se generan sin carpeta.

Nota Muy Importante: En este caso tenemos que escribir la ruta absoluta.

Nota: Véase el ejemplo ***AD04_JDBCHsqldb.java*** que ejecuta una consulta sobre ***Hsqldb***.

CONECTAR A H2 CON JDBC

En el fichero ***.zip*** que hemos descargado para la instalación de ***H2***, en la carpeta ***h2/bin***, se encuentra el ***driver JDBC*** para ***H2***: el fichero ***h2-1.4.196.jar***.

Hay que situar la librería en un lugar conocido (se recomienda ***/lib/h2***) e informar a ***Eclipse*** de que se encuentra ahí:

`CLICK-DCHO SOBRE EL PROYECTO/PROPIEDADES/VÍA DE CONSTRUCCIÓN JAVA/BIBLIOTECA DEL SISTEMA/AÑADIR JARS EXTERNOS`

Ahora el ***driver*** y el ***conector*** serán:

```
Class.forName("org.h2.Driver");
Connection conexion =
DriverManager.getConnection("jdbc:h2:RUTA_ABSOLUTA/nombreBD");
```

Nota Muy Importante: En este caso tenemos que escribir la ruta absoluta. Además el fichero que aloja la base de datos tiene la extensión ***.mv.db***, pero se escribe sin la extensión.

Nota: Véase el ejemplo ***AD05_JDBCH2.java*** que ejecuta una consulta sobre ***H2***.

2.5 Manipulación de Bases de datos relacionales

2.5.1 Sentencias DDL

DDL (Data Description Language) es la parte de ***SQL*** (Structured Query Language) que se encarga de la estructura de tablas y las relaciones entre sí. Lo habitual es que conozcamos la estructura de la BD con la que estamos trabajando, pero si no es así podemos obtener esta información a través de los llamados ***metadatos*** (datos sobre la estructura de la BD).

Los objetos más útiles para manejar estos ***metadatos*** son:

- ***DatabaseMetaData***: Tiene múltiple métodos para conocer el tipo de base de datos, el driver, la ***URL***, los nombres de usuarios, tablas y vistas. Podemos obtener este objeto a partir de la clase ***Connection***, con el método ***getMetaData()***.
- ***ResultSetMetaData***: Tiene métodos para conocer datos sobre una respuesta a consultas a la base de datos. Las consultas a la base de datos se responden en forma de tabla que, desde ***Java***, es un objeto ***ResultSet***. Aplicando el método ***getMetaData()*** a un objeto ***ResultSet*** obtenemos información sobre esa respuesta. Esa información vendrá en forma de objeto ***ResultSetMetaData***.

La clase ***DatabaseMetaData*** dispone (entre otros) de los siguientes métodos:

- ***getDatabaseProductName()***: Devuelve el nombre del ***SGBD*** (***MySQL***, ***H2***, etc).
- ***getDriverName():*** Devuelve el nombre del driver con el que se conecta.
- ***getURL()***: Devuelve la ruta (conector, ruta, carpeta) de la base de datos.
- ***getUserName()***: Devuelve el nombre del usuario conectado.

- ***getTables()***: Devuelve información sobre tablas y vistas de la base de datos. Necesita 4 parámetros de entrada:
  - ***Catálogo***: cuyos objetos queremos obtener. Con el valor ***null*** en este parámetro devolverá objetos de todos los catálogos.
  - ***Esquema***: cuyos objetos queremos obtener. El concepto de esquema puede ser distinto en cada ***SGBD***, por ejemplo, en ***MySQL*** tenemos un esquema por cada BD y en ***Oracle*** tenemos un esquema por cada usuario. El valor ***null*** en este campo hace referencia a todos los esquemas o al esquema actual según el ***SGBD***.
  - ***Patrón***: Usando los comodines guión bajo (_) y porcentaje (***%***), podemos filtrar los objetos que queremos por su nombre.
  - ***Tipo de objeto***: Para seleccionar tablas ***new String[]{"TABLE"}***, vistas ***new String[]{"VIEW"}***, ambos ***new String[]{"TABLE","VIEW"}***, o todos los tipos de objetos ***null***.

    El método ***getTables()*** devuelve la información en un objeto de la clase ***ResultSet***, que es una tabla más, de la que podemos obtener los datos a partir del orden o del nombre de la columna (las columnas están numeradas comenzando en 1).

.getTables()	
• TABLE_CAT	VARCHAR
• TABLE_SCHEM	VARCHAR
• TABLE_NAME	VARCHAR
• TABLE_TYPE	VARCHAR
• REMARKS	VARCHAR
• TYPE_CAT	VARCHAR
• TYPE_SCHEM	VARCHAR
• TYPE_NAME	VARCHAR
• SELF_REFERENCE_COL_NAME	VARCHAR
• REF_GENERATION	VARCHAR

Nota: Véase el ejemplo ***AD10_Metadatos1***.

- ***getColumns()***: Recibe como parámetros ***catálogo***, ***esquema***, nombre de ***tabla*** y nombre de ***columna*** y devuelve (en un ***ResultSet***) información sobre las columnas solicitadas. Tanto el nombre de tabla como el nombre de columna aceptan comodines.

  Nota: Véase el ejemplo ***AD11_Metadatos2***.

- ***getPrimaryKeys()***: Recibe como parámetros ***catálogo***, ***esquema*** y ***tabla*** y devuelve (en un ***ResultSet***) información sobre las columnas que son ***clave primaria*** en esa tabla.

  Nota: Véase el ejemplo ***AD12_Metadatos3***.

- ***getExportedKeys()***: Recibe como parámetros ***catálogo***, ***esquema*** y ***tabla*** y devuelve (en un ***ResultSet***) información sobre las columnas que son ***clave ajena*** en otra tabla. Esas columnas, naturalmente solamente pueden ser las primarias de esta tabla.

  Nota: Véase el ejemplo ***AD13_Metadatos4***.

- ***getImportedKeys()***: Recibe como parámetros ***catálogo***, ***esquema*** y ***tabla*** y devuelve (en un ***ResultSet***) información sobre las columnas que son clave ajena.

  Nota: Véase el ejemplo ***AD14_Metadatos5***.

- ***getProcedures()***: Recibe como parámetros ***catálogo***, ***esquema*** y ***procedimiento*** y devuelve (en un ***ResultSet***) información sobre el procedimiento indicado. Si el valor del procedimiento es ***null***, obtenemos información de todos los procedimientos de ese esquema.

  Nota: Véase el ejemplo ***AD15_Metadatos6***.

La clase ***ResultSetMetaData*** permite averiguar información sobre una tabla (***ResultSet***) y dispone (entre otros) de los siguientes métodos:

- ***getColumnCount()***: Devuelve el número de columnas de la tabla.
- ***getColumnName()***: Devuelve el nombre de la columna indicada por su índice.
- ***getColumnTypeName()***: Devuelve el tipo de dato de la columna indicada en el índice.
- ***isNullable()***: Devuelve ***0***, ***1*** ó ***2*** si la columna no permite valores nulos, los permite o no se sabe, respectivamente. La clase ***ResultSetMetaData*** tiene definidos los valores ***columNoNulls***, ***columnNullable*** y ***columnNullableUnknow***.
- ***getColumnDisplaySize()***: Devuelve el máximo ancho de caracteres de la columna.

  Nota: Véase el ejemplo ***AD16_Metadatos7***.

2.5.2 Sentencias DML

Ya conocemos el interfaz ***Statement*** (se genera con el método ***createStatement()*** de la clase ***Connection***) que nos permite crear consultas y ejecutarlas. ***Statement*** dispone de tres métodos:

- ***executeQuery()***: Se usa con consultas ***SELECT*** que recuperan datos y los devuelven en un ***ResultSet***.
- ***executeUpdate()***: Se usa con sentencias ***DDL***: ***CREATE***, ***DROP*** y ***ALTER*** y las sentencias ***DML***: ***INSERT***, ***UPDATE***, ***DELETE***. Este método devuelve un número entero que será ***0*** en el caso de las sentencias ***DDL*** y será el número de filas modificadas en el caso de las sentencias ***DML***.
- ***execute()***: Se usa con procedimientos almacenados en la base de datos y devuelven los datos en más de un ***ResultSet***.

Ejemplo: Inserción de un registro en la base de datos:

Nota: Este ejemplo está completo en ***AD17_InsertarPresi.java***.

Queremos insertar un registro en la tabla ***Presidentes***. El cuarto campo de esta tabla es la ***fecha de nacimiento*** que nuestra base de datos admite que sea un valor ***null***. La instrucción a ejecutar es la siguiente:

```
INSERT INTO Presidentes VALUES ('99999999I','Matías','Prats',null,'2019');
```

Desde ***Java*** preparamos la sentencia ***INSERT*** concatenando ***Strings***:

```
String consulta = "INSERT INTO Presidentes VALUES ('" + dni +
"','" + nombre + "','" + apellidos + "',null,'" + aAcceso +"')";
```

Nota: Observa que entre las comillas dobles de ***Java***, se usan comillas simples para ***MySQL***.

Posteriormente creamos una sentencia (***Statement***) y, con su método ***executeUpdate()***, ejecutamos nuestra sentencia ***INSERT***:

```
Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(consulta);
```

Nota: El método ***executeUpdate()*** devuelve el número de filas modificadas.

Ejemplo: Modificación de un registro en la base de datos:

Nota: Este ejemplo está completo en ***AD18_ModificarAforo.java***.

Queremos modificar el ***aforo*** de cada uno de los estadios añadiéndole ***50*** plazas a las que ya tenían. La instrucción a ejecutar es la siguiente:

```
UPDATE Equipos SET Aforo=Aforo + 50
```

Desde ***Java*** preparamos la sentencia ***UPDATE*** concatenando ***Strings***:

```
String sql = "UPDATE Equipos SET Aforo=Aforo + " + incremento;
```

Posteriormente creamos una sentencia (***Statement***) y, con su método ***executeUpdate()***, ejecutamos nuestra sentencia ***UPDATE***:

```
Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql);
```

Nota: Dispones de un ejemplo más para crear una vista MySQL en ***AD19_CrearVista.java***.

MARCADORES DE POSICIÓN

Como se puede ver en los ejemplos, la concatenación de ***String*** para construir las sentencias ***SQL*** puede ser engorrosa. Con las sentencias preparadas (***PreparedStatement***) podemos mejorar esta tarea mediante ***marcadores de posición*** que representan los datos que se asignarán posteriormente. Cada marcador de posición se representa con un interrogante (***?***), que son implícitamente numerados desde el uno en adelante.

Asignamos los marcadores con los métodos ***setTipo*** (***setInt()***, ***setString()***, etc) específicos para cada tipo de dato. El primer argumento es el número de marcador, el segundo el dato.

Ejemplo: Inserción de un registro en la base de datos (con consulta preparada):

Nota: Este ejemplo está completo en ***AD17B_InsertarPresi_Preparada.java***.

```
String select = "INSERT INTO Presidentes VALUES ( ?, ?, ?, null, ?)";
// Preparamos la sentencia
PreparedStatement sentencia = conexion.prepareStatement(select);
// Sustituimos cada marcador con su dato
sentencia.setString(1, dni);
sentencia.setString(2, nombre);
sentencia.setString(3, apellidos);
sentencia.setInt(4, Integer.parseInt(aAcceso));
```

Los marcadores de posición solamente pueden ocupar sitio de datos, no de tablas o columnas. Después de asignar los marcadores con datos se puede ejecutar la sentencia.

Nota: Los objetos ***Statement*** necesitan que aportemos la sentencia ***SQL*** cada vez que los queramos ejecutar. Los objetos ***PreparedStatement***, en cambio, se preparan una sola vez y pueden ejecutarse tantas veces como queramos.

Ejemplo: Modificación de un registro en la base de datos:

Nota: Este ejemplo está completo en ***AD18B_ModificarAforo_Preparada.java***. El mismo ejemplo sin marcadores de posición lo puedes ver en ***AD18_ModificarAforo.java***.

```
String sql = "UPDATE Equipos SET Aforo = Aforo + ?";
PreparedStatement sentencia = conexion.prepareStatement(sql);
sentencia.setInt(1, incremento);
```

Nota: También es posible ejecutar sentencias ***SELECT*** con ***sentencias preparadas***. Véase el ejemplo ***AD20_SELECTPreparado.java***.

2.5.3 Procedimientos

Las bases de datos relacionales admiten la inclusión de programas que pueden recibir argumentos de entrada y pueden devolver o no un valor. Cuando devuelven un valor se llaman ***funciones*** y cuando no lo hacen se llaman ***procedimientos***.

Ejemplo: procedimiento fichaje para MySQL:

```
DELIMITER //

CREATE PROCEDURE fichaje (j char(5), e char(5), goles int)
BEGIN
  DECLARE golesConseguidos integer default 0;
  SELECT count(*) INTO golesConseguidos FROM Goles WHERE Cod_Jugador=j;
  IF (golesConseguidos >= goles)
    THEN UPDATE Jugadores SET Cod_Equipo = e WHERE Codigo=j;
  END IF;
END;
//

DELIMITER ;
```

Nota: Con ***DELIMITER*** modificamos el delimitador de final de sentencia. Habitualmente cada sentencia acaba con un punto y coma (***;***). Cuando creamos un procedimiento, necesitamos un delimitador distinto para no dar por terminado el procedimiento con cada punto y coma de las sentencias internas al procedimiento. Al finalizar restablecemos el delimitador habitual.

En este ejemplo vemos un procedimiento llamado ***fichaje*** que recibe tres parámetros: un jugador (***j***), un equipo (***e***) y un número de goles (***goles***).

Contamos los goles conseguidos por el jugador. Si igualan o exceden el número de goles indicado, el jugador fichará por el equipo (es decir, modificamos el ***Cod_equipo*** en el registro del jugador).

Ejemplo: procedimiento fichaje para Oracle:

```
CREATE OR REPLACE PROCEDURE fichaje
  (j Jugadores.Codigo%TYPE, e Equipos.Codigo%TYPE, goles NUMBER) AS
  golesConseguidos NUMBER;
BEGIN
  SELECT count(*)
  INTO golesConseguidos
  FROM Goles
  WHERE Cod_Jugador = j;

  IF (golesConseguidos >= goles)
    THEN UPDATE Jugadores SET Cod_Equipo = e WHERE Codigo=j;
  END IF;
END;
/
```

Ejemplo: función golesEquipo para MySQL:

```
DELIMITER //

CREATE FUNCTION golesEquipo (e char(5)) RETURNS integer
BEGIN
    DECLARE nGoles integer default 0;
    SELECT count(*)
    INTO nGoles
    FROM Goles, Jugadores
    WHERE Goles.Cod_Jugador = Jugadores.Codigo
          AND Jugadores.Cod_Equipo = e;
    RETURN nGoles;
END;
//

DELIMITER ;
```

La función ***golesEquipo*** cuenta todos los goles del equipo que recibe como parámetro y los devuelve en un entero. Observa ***RETURNS integer*** en la cabecera de la función y ***RETURN nGoles;*** al final.

Ejemplo: función golesEquipo para Oracle:

```
CREATE OR REPLACE FUNCTION golesEquipo (e Equipos.Codigo%TYPE)
    RETURN NUMBER AS
    nGoles NUMBER;
BEGIN
    SELECT count(*)
    INTO nGoles
    FROM Goles, Jugadores
    WHERE Goles.Cod_Jugador = Jugadores.Codigo
          AND Jugadores.Cod_Equipo = e;

    RETURN nGoles;
END;
/
```

LLAMADA A PROCEDIMIENTOS Y FUNCIONES DESDE JAVA

Para poder usar desde ***Java*** los ***procedimientos***/***funciones*** almacenados en la base de datos, usamos el método ***prepareCall()*** de la clase ***Connection*** que devuelve una sentencia ejecutable (***CallableStatement***). El método ***prepareCall*** recibe un ***String*** con la llamada al ***procedimiento***/***función***. Este ***String*** puede contar también con marcadores de posición, incluso en el valor que devuelve una función.

Ejemplo: Llamada al procedimiento fichaje:

Nota: Ejemplo completo en ***AD21A_ProcFichajeMySQL*** y ***AD21B_ProcFichajeOracle***.

```
String sql = "{ call fichaje (?, ?, ?) } ";
// Preparamos la llamada
CallableStatement llamada = conexion.prepareCall(sql);
llamada.setString(1, jugador);
llamada.setString(2, equipo);
llamada.setInt(3, goles);
// Ejecutamos el procedimiento
llamada.executeUpdate();
```

En este código cada uno de los marcadores de posición se asocia con una variable a través de los métodos ***setString()*** o ***setInt()***.

Ejemplo: Llamada a la funcion golesEquipos:

Nota: Ejemplo completo en ***AD22A_FuncNombreMySQL*** y ***AD22B_FuncNombreOracle***.

```
String sql = "{ ? = call golesEquipo (?) } ";
// Preparamos la llamada
CallableStatement llamada = conexion.prepareCall(sql);
// Establecemos los marcadores y registramos la salida
llamada.registerOutParameter(1, Types.INTEGER);
llamada.setString(2, equipo);
```

En este ejemplo, el primer marcador de posición no es un parámetro de entrada, sino la salida de función que devolverá un número entero. Observa como se registra el ***marcador de posición número 1*** como un ***parámetro de salida*** de tipo entero.

El marcador de posición número 2 es un parámetro de entrada y se asocia con una variable a través del método ***setString()***.

Una vez ejecutada la función se pueden obtener los datos de los parámetros de salida con los métodos ***getTipo(indice)*** específicos para cada tipo de dato:

```
int goles = llamada.getInt(1);
```

PARÁMETROS DE SALIDA

Según el ***SGBD*** los argumentos de un ***procedimiento***/***función*** pueden ser de entrada (***IN),*** de salida (***OUT***) y de entrada/salida (***INOUT***) como ocurre en ***Oracle***, o bien, los parámetros son siempre de entrada como ocurre en ***MySQL***.

Cuando los ***procedimientos***/***funciones*** tienen parámetros de salida hay que registrarlos antes de la llamada con el método ***registerOutParameter(posición, tipo)*** (de la misma forma que hemos

hecho con el valor de salida de la función en el último ejemplo). La posición es la del argumento como marcador de posición y el tipo es uno de los registrados en ***java.sql.Types*** que recoge los tipos usados en ***SQL***.

Ejemplo: Procedimiento Oracle con parámetros de salida y llamada desde Java

Nota: Ejemplo completo en ***AD23_DivisionEntera.java***.

```
CREATE OR REPLACE PROCEDURE divisionEntera
    (dividendo IN NUMBER, divisor IN NUMBER,
     cociente OUT NUMBER, resto OUT NUMBER) AS

BEGIN
    cociente := trunc(dividendo/divisor);
    resto := dividendo - (cociente*divisor);
END;
/
```

En la cabecera del procedimiento indicamos para cada parámetro si es de entrada (***IN***), si es de salida (***OUT***) o si es de entrada/salida (***INOUT***).

```
String sql = "{ call divisionEntera (?, ?, ?, ?) } ";
// Preparamos la llamada
CallableStatement llamada = conexion.prepareCall(sql);
llamada.setInt(1, dividendo);
llamada.setInt(2, divisor);
llamada.registerOutParameter(3, Types.INTEGER);
llamada.registerOutParameter(4, Types.INTEGER);
// Ejecutamos e imprimimos el resultado
llamada.executeUpdate();
System.out.println("Al dividir "+dividendo+" y "+divisor);
System.out.println("\tobtenemos "+llamada.getInt(3)+" y resto
"+llamada.getInt(4));
```

En el código ***Java*** asociamos cada marcador de posición con una variable de entrada, o bien, lo registramos como un parámetro de salida que, posteriormente, recogemos con ***getInt()*** (o ***getString()***, etc).

2.5.4 Gestión de errores SQL

La excepción ***SQLException*** dispone de varios métodos que nos ayudan a localizar los errores y que conviene usar en lugar del habitual ***printStackTrace()***. Esos métodos son:

- ***getMessage()***: Mensaje descriptivo del error.
- ***getSQLStatus()***: Estado según la definición del estándar X/OPEN SQL
- ***getErrorCode()***: Código de error del fabricante que devuelve la propia base de datos.

Por tanto, sería conveniente cazar esta excepción con el siguiente código:

```
catch (SQLException sqlE) {
    System.err.println("\nHubo problemas con la base de datos:");
    System.err.println("Mensaje:\t"+sqlE.getMessage());
    System.err.println("Estado SQL:\t"+sqlE.getSQLState());
    System.err.println("Codigo Error:\t"+sqlE.getErrorCode());
}
```

2.6 Bases de datos no relacionales embebidas

- ***DB4O***
- ***Oracle Berkeley***

2.6.1 DB4O (Database for Object)

Es un gestor de bases de datos orientadas a objetos disponible para ***Java*** y ***.Net***. No hay desfase objeto-relacional. No usa ***SQL*** y la ***BD*** tiene la extensión ***.YAP***.

Se instala haciendo uso de una librería ***.jar*** (para ***Java***) o ***.dll*** (para ***.NET***).

- Web de descarga: http://www.db4o.com
- Fichero: ***db4o-8.0.276.16149-java.zip***

Al descomprimir aparece una carpeta ***/lib*** donde se encuentran los ***.jar*** (***db4o-XXXX.jar***) que tendremos que añadir a nuestro proyecto.

Para usar desde ***Eclipse***:

`CLICK-DCHO SOBRE EL PROYECTO/PROPIEDADES/VÍA DE CONSTRUCCIÓN JAVA/BIBLIOTECA DEL SISTEMA/AÑADIR JARS EXTERNOS`

En realidad no es necesario añadir todos los ***.jar***, bastaría con ***db4o-X.X.XXX.XXXXX-core-java5.jar***.

Nota: Al incorporar las librerías indicamos la ruta, y ***Java*** las buscará siempre en esa ruta, es decir, no quedan incorporadas. Por eso es conveniente buscarles un sitio adecuado y permanente, dejar allí una copia y usar esa ruta. Un buen sitio sería: ***/lib/db4o***.

USO DE DB4O DESDE JAVA

Para crear la BD (o acceder a una BD ya creada):

```
ObjectContainer db =
Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "DB.yap");
```

El nombre del fichero tendrá la extensión ***.yap***.

Añadir objetos a la BD:

```
db.store(empleado);
```

Eliminar objetos a la BD:

```
db.delete(empleado);
```

Cerrar la BD:

```
db.close();
```

Para recuperar objetos de la BD (una consulta):

```
respuesta = db.queryByExample(empleado);
```

donde el objeto que pasamos (***empleado***) debe tener inicializados los atributos a emparejar.

Ejemplo: Recuperar todos los empleados cuyo nombre sea Adolfo.

```
Empleado empleado = new Empleado();
empleado.setNombre("Adolfo");
respuesta = db.queryByExample(empleado);
```

En este caso hemos creado un nuevo ***empleado*** con el constructor por defecto, de modo que tiene todos los atributos sin inicializar. Posteriormente solamente asignamos valor al ***nombre***. Este objeto nos sirve de patrón para la consulta con el método ***queryByExample()***.

Ejemplo: Recuperar todos los empleados.

```
Empleado empleado = new Empleado();
respuesta = db.queryByExample(empleado);
```

En este caso hemos creado un nuevo ***empleado*** con el constructor por defecto, de modo que tiene todos los atributos sin inicializar. Este objeto nos sirve de patrón para la consulta con el método ***queryByExample()***. Como no hemos establecido ninguno de los atributos encaja con todos los empleados.

El método ***queryByExample()*** devuelve un objeto de la clase ***ObjectSet*** (conjunto de objetos) que podemos manipular con los siguientes métodos:

- ***size()***: Devuelve el número de objetos del conjunto.
- ***hasNext()***: Devuelve si quedan (***true***) o no (***false***) objetos para leer.
- ***next()***: Devuelve el siguiente objeto del conjunto.

Para modificar un objeto de la BD se localiza, se modifica y se vuelve a almacenar:

```
Empleado empleado = (Empleado) respuesta.next();
empleado.setEdad(edad);
db.store(empleado);
```

Nota: Véase el ejemplo ***AD08_DB4O.java***, junto con ***AD08_Empleado.java***.

2.7 Patrón Modelo-Vista-Controlador (MVC)

2.7.1 Conceptos previos

Javabean: componente software reutilizable. Se trata de una ***clase Java*** que tiene estas características:

- Un constructor sin argumentos, aunque puede tener otros constructores adicionales.
- Sus atributos son privados.
- Todos los atributos tienen los métodos ***get*** y ***set*** con la nomenclatura estándar.
- Debe ser ***serializable***, es decir, implementar la interfaz ***java.io.Serializable***. Un objeto serializable es un objeto que se puede transformar en una cadena de bits para poder ser transferido a través de un ***Stream***.

POJO: (Plain Old Java Object): es una ***clase Java*** muy simple, con ***contructor***, ***getters*** y ***setters***, pero que no extiende ni implementa otras clases.

Servlet: Es una clase ***Java*** (extiende de ***HttpServlet***) que amplía la funcionalidad de un servidor web. Cada servlet se asocia con una localización del servidor (una ***URL***) y dispone de un método ***service()*** que se ejecuta cuando se hace una petición a esa ***URL***. El método ***service()*** puede recoger parámetros adicionales de la ***URL***. Una servlet dispone además de otros métodos: ***init()*** que se ejecuta la primera vez que el servlet es invocado y el método ***destroy()*** que se ejecuta al final para eliminar el servlet y liberar sus recursos.

El método ***service()*** de la clase ***HttpServlet*** tiene dos argumentos:

```
protected void service(HttpServletRequest peticion, HttpServletResponse respuesta)
```

- ***HttpServletRequest***: Representa el objeto de la petición que se ha hecho solicitando que se ejecute este recurso. Contiene entre otras cosas, los argumentos con los que se hizo la llamada.
  - El método ***getParameter("nombre")***: lo usamos para obtener los parámetros explícitos en la ***URL***.
  - El método ***setAtribute("nombre", objeto)***: es para anexar cualquier objeto a la petición. Es posible que desde este recurso queramos reenviar la petición a otro recurso. Con este método podemos añadir contenido antes de reenviarlo.
  - El método ***getAttribute("nombre")***: nos vale para recuperar esos objetos anexados.
- ***HttpServletResponse***: Representa la respuesta ***HTTP***.
  - El método ***setRedirect("fichero.jsp")***: permite redirigir el flujo a otro recurso, tal como un ***JSP*** o un ***HTML***.

Nota: La asociación entre una ***URL*** y la ***servlet*** que atenderá la petición se hace en un fichero ***.xml*** (***web.xml***) que se denomina descriptor de despliegue de la aplicación.

Dentro de una servlet también podemos necesitar objetos ***RequestDispatcher*** que representan a cualquier recurso que puede recibir y procesar peticiones del cliente (como una ***servlet***, un ***HTML*** o un ***JSP***). Una de las utilidades de la clase ***RequestDispatcher*** es que dispone de un método ***forward()*** que permite reenviar la petición recibida (manipulada o no) a otro recurso.

```
// Obtenemos la lista de presidentes
ArrayList<Presidente> listaPresidentes = gestor.listaPresi();
// La anexamos a la petición recibida
peticion.setAttribute("presidentes", listaPresidentes);
// Obtenemos una referencia al JSP listaPresi.jsp
RequestDispatcher rd =
this.getServletContext().getRequestDispatcher("/listaPresi.jsp");
// Reencaminamos la petición recibida (con la lista anexada) al JSP
rd.forward(peticion, respuesta);
```

En este ejemplo añadimos un atributo a la petición recibida (método ***setAttribute()***) y posteriormente lo reenviamos (método ***forward()***) a otro recurso, en este caso a un ***JSP*** llamado ***listaPresi.jsp***.

Applet: Es una clase ***Java*** (extiende de ***Applet***) que se ejecuta en un navegador web que tenga instalado el plugin de ***Java***. Actualmente los navegadores más extendidos no aceptan este plugin porque las applets pueden contener software malicioso.

JSP: (Java Server Pages) Las páginas ***JSP*** permiten generar contenido dinámico en la web. Son documentos ***HTML*** o ***XML*** que incrustan código ***Java*** (***scriptlets***). Son una alternativa a ***PHP*** como lenguaje de script de servidor.

El código incrustado se encierra entre los caracteres <% y %>.

2.7.2 Elementos de JSP

En una página ***JSP*** podemos encontrar, además de ***HTML***, los siguientes elementos:

- ***Directivas***: <%@ ... %>
  - ***include***: Para incluir un fichero mediante el atributo ***file***:

```
<%@ include file="fichero.html" %>
```

  - ***page***: Para indicar características de la página. Usa atributos como ***language***, ***pageEncoding***, ***import***, etc:

```
<%@ page language="java" pageEncoding="UTF-8" import="java.util.*"%>
```

- ***Declaraciones***: <%! ...%> Para declarar variables y funciones:

```
<%! int maxValor = 30; %>
```

- ***Scriptlets***: <% *código Java* %> Son trozos de código ***Java*** incrustado entre los elementos ***HTML***:
- ***Expresiones***: <%= *expresión Java* %> Se evalúan y devuelven un valor (no se acaban en punto y coma (;).

```
<TD><%=presi.getDni()%></TD>
```

  En este ejemplo se obtiene el valor del método ***getDni()*** y se escribe dentro de la celda de una tabla ***HTML***.

- ***Etiquetas JSP***: Proporcionan una funcionalidad básica y permiten trabajar con componentes complementarios como ***applets***, otras páginas ***JSP***, ***Javabeans***, etc.
  - No asociadas a ***Javabeans***:
    - ***<jsp:forward> ... </jsp:forward>*** ó ***<jsp:forward ... />***: Redirige la petición a otro recurso:

```
<jsp:forward page="/principal?comando=nuevoPresi"/>
```

    - ***<jsp:include> ... </jsp:include>*** ó ***<jsp:include ... />***: Incluye el texto de un fichero dentro de la página.
    - ***<jsp:plugin> ... </jsp:plugin>***: Descarga un ***applet*** o un ***bean*** de ***Java***.
  - Asociadas a ***Javabeans***:
    - ***<jsp:useBean> ... </jsp:useBean>*** ó ***<jsp:useBean ... />***: Permite usar un ***bean***.

```
<jsp:useBean id="presi" scope="request" class="liga.Presidente"/>
```

Esta etiqueta tiene 3 atributos: ***id*** para nombrar el ***bean***, ***class*** para indicar la clase del ***bean*** y ***scope*** para indicar su ámbito. El ámbito es la zona de alcance donde será visible el ***bean***, que puede ser ***page***, ***application***, ***session*** o ***request***. Si dentro de ese ámbito se usa otro ***bean*** con el mismo nombre, solo será instanciado si no existía ya (el código entre apertura y cierre de ***useBean*** sólo se ejecuta si se instancia ahora, no si ya existía). Un ámbito ***request*** asocia el ***bean*** a la petición que ha llegado a la página ***JSP*** (variable implícita ***request***) y lo incluye en esa petición como si se ejecutara ***request.setAttribute("nameBean",bean)***;

- ***<jsp:setProperty ... />***: Establece el valor de un atributo de un ***bean***.

Esta etiqueta tiene 3 atributos: ***name*** para indicar el ***bean***, ***property*** para indicar el ***atributo de clase*** y ***value*** para el nuevo ***valor***.

El valor que se asigna al atributo de clase se puede indicar con ***param*** en lugar de ***value***. En ese caso se indica el nombre del parámetro ***HTML*** desde el que hay que recoger el valor. Si el nombre del parámetro coincide con el de la propiedad, lo podemos omitir.

Si los parámetros ***HTML*** se llaman igual que los atributos de clase del ***bean***, se pueden asignar todos de una vez indicando ***property="*"***. Un asterisco en el atributo ***property*** hace referencia a todos los atributos de clase del ***Bean***.

Nota: Puedes ver un ejemplo explicado en el fichero ***altaPresi.jsp*** de la aplicación ***ADT2_Liga***.

- ***<jsp:getProperty ... />***: Obtiene un atributo de un ***bean***.

Esta etiqueta tiene 2 atributos: ***name*** para indicar el ***bean***, ***property*** para el a***tributo de clase*** del ***bean***.

- ***Comentarios JSP***:

```
<%-- comentario JSP --%>
```

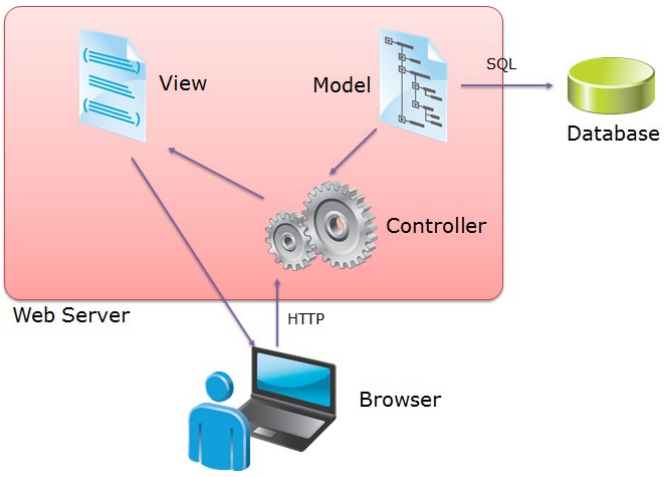
- ***Variables implícitas***: Las páginas ***JSP*** incluyen unas variables que pueden ser usadas en el código ***Java*** de la página sin declarar ni inicializar, algo así como se usan los argumentos de un método en su interior, pero no existe una cabecera de método donde puedas verlas. Estas variables son, entre otras:
  - ***request***: (del tipo ***HttpServletRequest***), es la petición con la que se ha solicitado el recurso.
  - ***response***: (del tipo ***HttpServletResponse***), es la respuesta ***HTTP*** a la solicitud.
  - Existen otras. En https://es.wikipedia.org/wiki/JavaServer_Pages puedes ver la lista completa.

En http://chuwiki.chuidiang.org/index.php?title=Pasar_datos_entre_JSPs_y_Servlets._Page,_Request,_Session_y_Application_scope hay un resumen de como comparten datos los ***JSP*** y los ***servlet***.

2.7.3 ¿Qué es el patrón MVC?

El patrón Modelo-Vista-Controlador (***MVC***) es una estrategia de programación que resulta adecuada en aplicaciones con una fuerte interacción con el usuario. Este patrón organiza la aplicación en tres bloques:

- El ***modelo*** es la estructura de datos y se relaciona con la BD.
- La ***vista*** es la parte que interactúa con el usuario, como los formularios de entrada y los informes de salida.
- El ***controlador*** es el programa subyacente que controla todo, recibe los datos del usuario, interpreta sus peticiones, recoge los datos del ***modelo*** y encarga que los muestre a la ***vista***.



Las ventajas de este patrón son muchas:

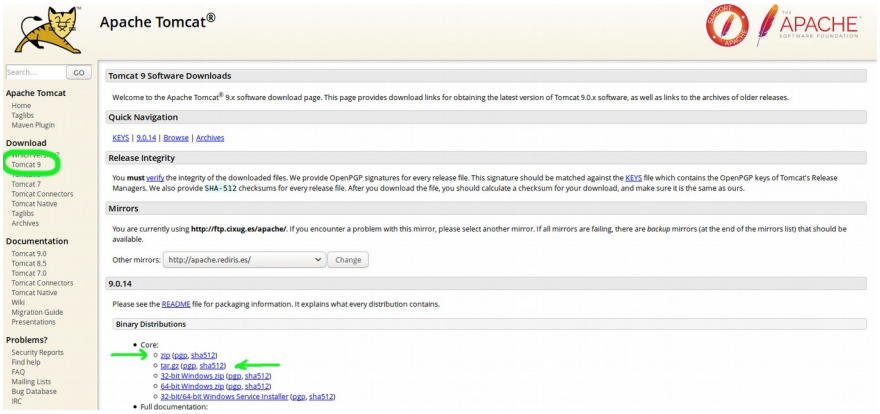
- Separa los conceptos facilitando el desarrollo de aplicaciones modulares en paralelo.
- Reutilización del código.
- Facilita las pruebas y el mantenimiento.

2.7.4 Tomcat

Tomcat es un servidor web con soporte para ***servlets*** y ***JSP***. Está integrado por el servidor web ***Apache*** y el contenedor de servlets ***Catalina***.

Podemos descargar ***Tomcat*** desde su página oficial en http://tomcat.apache.org. La última versión en el momento de escribir este documento es la ***9.0.14***, así que seleccionamos la ***versión 9*** en la izquierda (ver marcas verdes en la imagen) y seleccionamos el fichero ***zip*** para ***Windows*** o el fichero ***tar.gz*** para ***Linux***.

Una vez descargado el fichero ***apache-tomcat-9.0.14.tar.gz*** (o ***apache-tomcat-9.0.14.zip*** para ***Windows***), la instalación consiste en descomprimir el fichero en el sitio adecuado.



En el caso de ***Ubuntu*** se recomienda extraerlo en ***/opt***:

```
cd /opt
sudo tar -xzvf rutaFichero/apache-tomcat-9.0.14.tar.gz
```

Se genera una carpeta ***apache-tomcat-9.0.14***, que en este documento llamaremos ***rutaTomcat***.

La carpeta de ***Tomcat*** que se ha creado es propiedad del usuario ***root***. Será necesario modificar su propietario al usuario que usamos habitualmente:

```
sudo chown -R admin:admin /opt/apache-tomcat-9.0.14
```

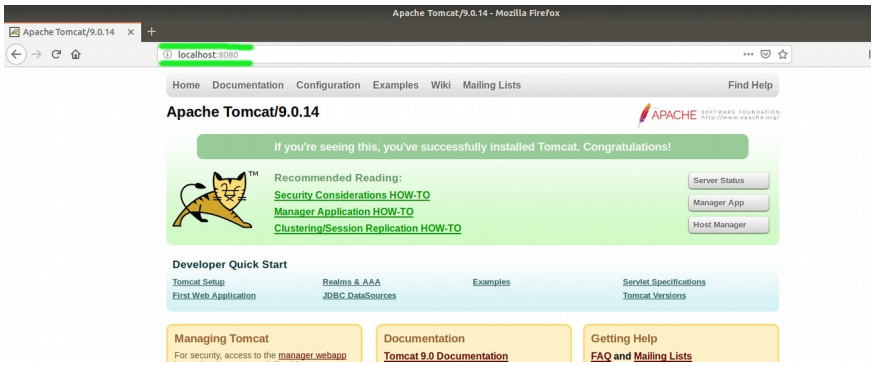
sustituyendo ***admin*** por tu usuario habitual.

Para iniciar el servidor basta con ejecutar ***startup.sh*** (en ***Linux***) o ***startup.bat*** (en ***Windows***), que se encuentran en la subcarpeta ***rutaTomcat/bin***.

```
sudo sh /opt/apache-tomcat-9.0.14/bin/startup.sh
```

Nota: Como siempre, podemos incluir la ruta del fichero ***.sh*** en la variable de entorno ***PATH***.

Una vez el servidor ***Tomcat*** levantado, desde el navegador, hacemos una petición ***HTTP*** al puerto ***8080*** del equipo local (MUY IMPORTANTE, el servidor ***Tomcat*** por defecto escucha en el puerto ***8080***) y aparecerá la página de bienvenida, lo que es señal de que la instalación es correcta.



Para detener el servidor usamos los scripts ***shutdown.sh*** (***Linux***) o ***shutdown.bat*** (***Windows***).

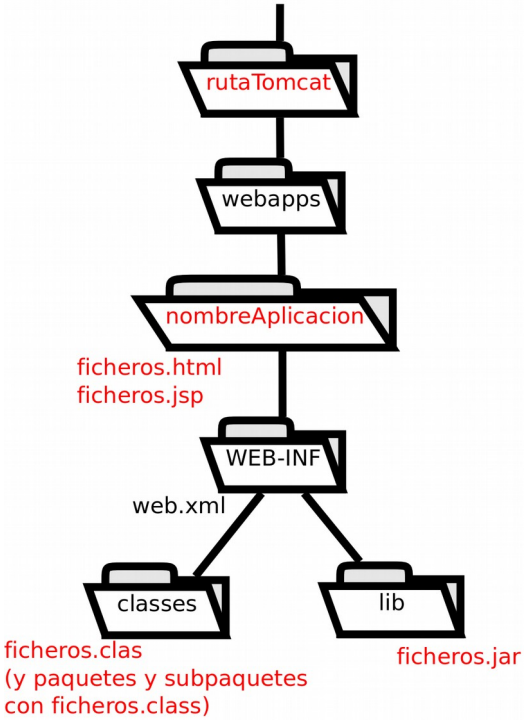
```
sudo sh /opt/apache-tomcat-9.0.14/bin/shutdown.sh
```

2.7.5 Java Enterprise Edition (J2EE)

Hay muchos frameworks que trabajan con este patrón, entre ellos ***J2EE*** (Java Enterprise Edition), que permite crear aplicaciones con una arquitectura llamada ***Modelo 2***. Para implementar esta arquitectura necesitamos un contenedor de ***Servlets*** como ***Tomcat***. En la arquitectura ***Modelo 2*** tenemos los siguientes elementos:

- El ***modelo*** mediante ***Javabeans*** o ***POJOs*** (Plain Old Java Object).
- La ***vista*** mediante ***JSP*** para la interfaz de usuario. Las páginas ***JSP*** usan los ***Javabeans*** como componente modelo.
- El ***controlador*** mediante clases ***Java***.

La aplicación estará formada por un conjunto de ficheros ***JSP***, ***HTML***, ***XML***, ***Java***, librerías ***JAR*** con la estructura que se muestra en la imagen de la derecha.



Debes tener en cuenta que todos los nombres de carpeta o fichero que aparecen en color negro en la imagen deben escribirse exactamente igual, incluida la combinación de mayúsculas/minúsculas.

En la carpeta en la que hemos realizado la instalación de ***Tomcat*** (***rutaTomcat***) hay una subcarpeta ***webapps*** para las aplicaciones. Aquí es donde debemos colocar nuestra aplicación: una carpeta con los ficheros ***HTML*** y ***JSP*** y una subcarpeta llamada ***WEB-INF*** con el descriptor de despliegue (llamado ***web.xml***) y las subcarpetas ***classes*** y ***lib***, para las clases y librerías ***Java*** respectivamente.

Las clases se pueden agrupar en paquetes y subpaquetes, lo que desde el punto de vista del sistema de archivos serán carpetas y subcarpetas.

En la carpeta de la aplicación siembre habrá una carpeta llamada ***WEB-INF*** (en mayúsculas) con el siguiente contenido:

- Ficheros ***HTML*** ó ***JSP***: Necesarios para la aplicación.
- Fichero ***web.xml***: Es el descriptor de despliegue de la aplicación. Se trata de un fichero ***XML*** que nos generará el propio ***IDE*** para que tenga la estructura y cabecera adecuada. Dentro de ese descriptor prestaremos atención a dos cosas:
  - ***ficheros por defecto***: Son los nombres de fichero que se buscan cuando se solicita una ***URL*** sin indicar el fichero (lo que en ***Apache*** se hace con la directiva ***DirectoryIndex***). Un ejemplo sería:

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

En este ejemplo, cuando en el navegador se escriba una ***URL*** sin indicar nombre de fichero, se buscará en orden: ***index.html*** o ***index.htm*** o ***index.jsp***.

- ***mapeo de servlets***: Por cada servlet de nuestra aplicación necesitamos incluir dos elementos: <***servlet-mapping***> para asociar el nombre de servlet con una localización (***URL***) y <***servlet***> para asociar el nombre de servlet con la ***clase Java*** que la contiene. Ejemplo:

```
<servlet-mapping>
  <servlet-name>Altas</servlet-name>
  <url-pattern>/insertar</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>Altas</servlet-name>
  <servlet-class>Alta</servlet-class>
</servlet>
```

En este ejemplo nuestra aplicación tiene una servlet que llamamos ***Altas***, que será invocada en la localización ***/insertar***, es decir, cuando en el navegador se solicite la ***URL***: ***http://dominioServidor/insertar***.

Por otro lado vemos que esa servlet llamada ***Altas*** está asociada a la clase Java ***Alta***. Por tanto cuando se pida la ***URL*** se ejecutará el método ***service()*** de la clase ***Alta***.

Nota: Es habitual que el nombre de la ***servlet***, la ***localización*** y la ***clase*** tengan el mismo nombre.

- Carpeta ***classes***: con los ficheros ***.class*** distribuidos en paquetes y subpaquetes. Aquí es donde estarán los ***Servlets*** y ***Javabeans*** que son, en definitivas, clases ***Java***.
- Carpeta ***lib***: con las librerías ***JAR*** que necesite la aplicación.

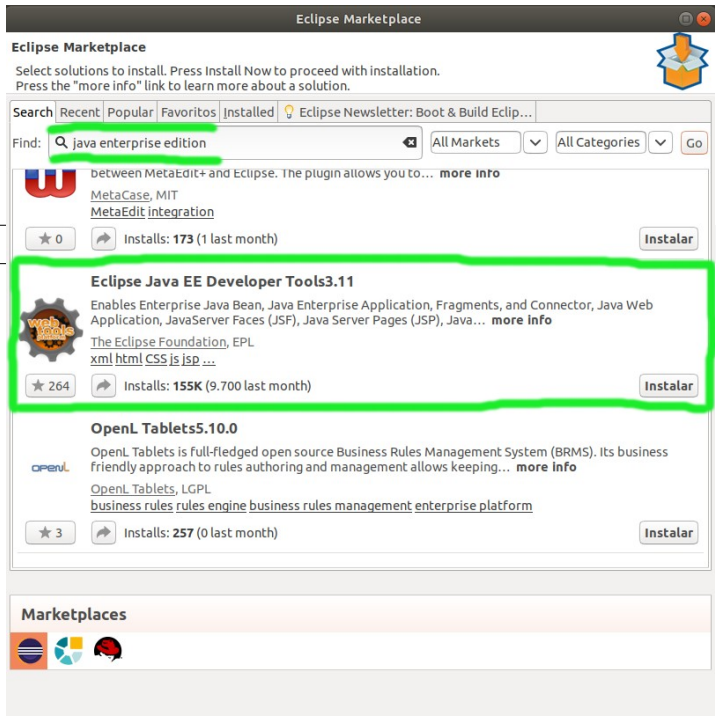
2.7.6 Preparar Eclipse para J2EE

Si cuando instalaste ***Eclipse*** elegiste el modelo ***Eclipse for J2EE Developers***, no tendrás que añadir nada. Si no fue así, simplemente hemos de añadir un plugin desde:

Ayuda/Eclipse Marketplace

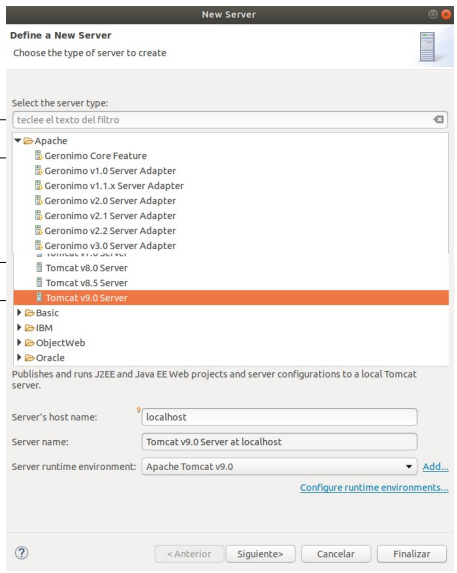
y buscar ***Eclipse Java EE Developer Tools 3.11*** como en la imagen de la derecha.

Recuerda que la descarga e instalación de plugins en ***Eclipse*** se hace en segundo plano y, aunque parezca haber terminado, has de esperar a que te pida reiniciar el programa.



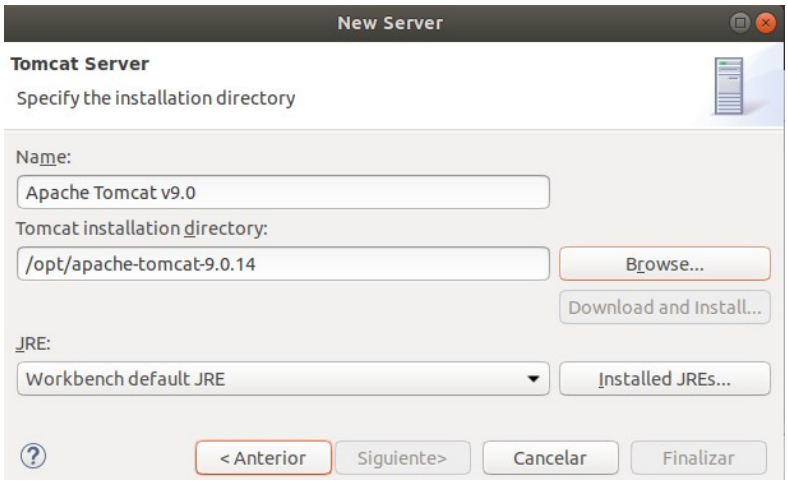
Una vez terminada la instalación y reiniciado ***Eclipse*** hemos de añadir el servidor que queremos usar. Puedes acceder a la ventana de servidores desde el menú:

Ventana/Mostrar vista/Servidores



Un clic-dcho te permitirá añadir un nuevo servidor, en nuestro caso elegimos:

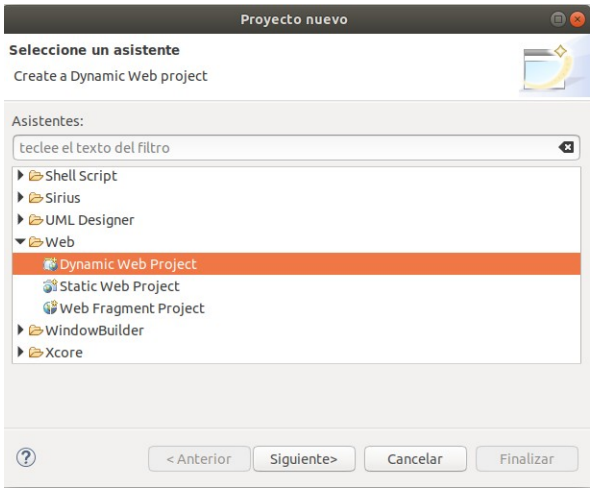
Apache/Tomcat v9.0 Server



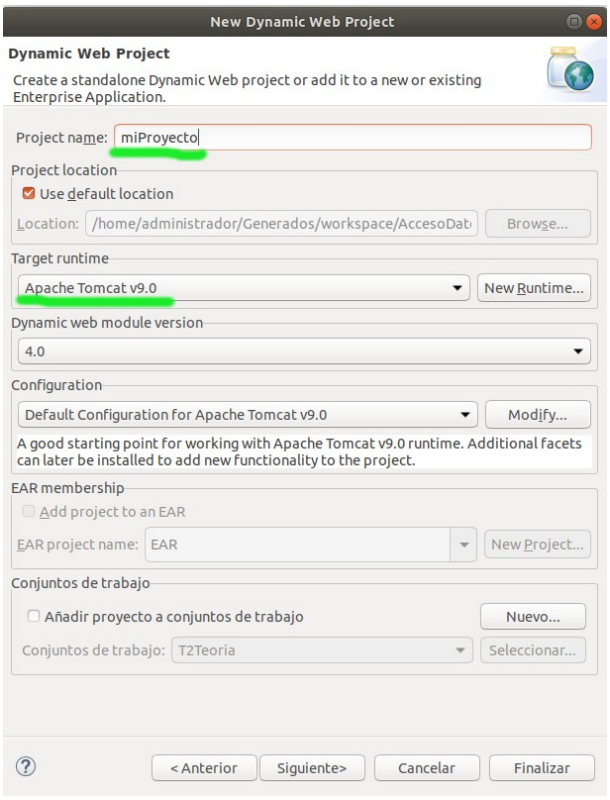
Crear un proyecto Java para web

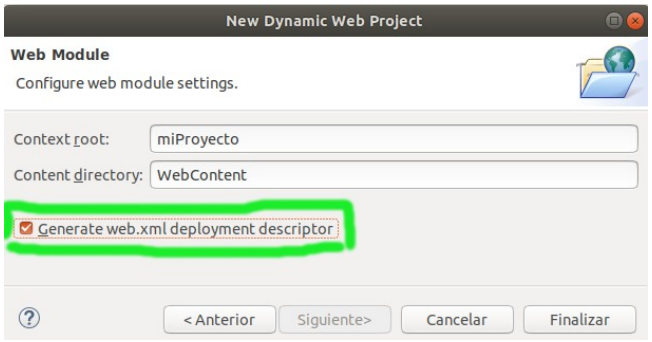
Nos desplazamos a:

Archivo/Nuevo Proyecto/Web/Dinamic Web Project



Indicamos el nombre del proyecto y el entorno de ejecución que, en nuestro caso, es la ***versión 9 de Apache Tomcat.*** Pasamos por las siguientes pantallas sin hacer cambios, pero al final será necesario marcar la casilla para que genere el descriptor de proyecto, es decir, el fichero, ***web.xml***.





Probablemente veamos un mensaje indicando que el proyecto recién creado está asociado con otra perspectiva. Lo aceptamos.

Como podemos ver, este proyecto tiene más carpetas que un proyecto ***Java*** simple. Es importante que coloquemos cada elemento en el lugar adecuado:

- Los ficheros fuente ***Java***: en la carpeta ***src*** (source) de ***Java Resources***.
- Los fichero ***HTML*** y ***JSP***: en la carpeta ***WebContent***.
- El fichero ***web.xml***: en la carpeta ***WebContent/WEB-INF*** (si elegimos crearlo con el proyecto ya aparecerá ahí).
- Las librerías ***Java*** (ficheros ***JAR***): en la carpeta ***WebContent/WEB-INF/lib***

GENERAR Y DESPLEGAR LA APLICACIÓN

Una vez terminada la aplicación, la exportamos con:

```
Archivo/Exportar/Web/Archivo WAR
```

e indicamos el destino. Verás que se ha generado un fichero ***.war*** que comprime toda la aplicación. Para desplegarla solamente tenemos que llevarla a la carpeta:

```
rutaTomcat/webapps
```

Ni siquiera es necesario descomprimir.

Ahora podremos ver la aplicación, desde un navegador, con la ***URL***:

```
http://dominioServidor:8080/nombreAplicacion
```