FACULTAD DE INGENIERÍA UNAM
DIVISIÓN DE EDUCACIÓN CONTINUA

Curso a Distancia

Características Avanzadas del Lenguaje JAVA

Material Didáctico

Enero 16, 2006
Ing. Dan Schmidt Valle
CAD 36

I. JDBC

1. Introducción

JDBC (*Java DataBase Connectivity*) es un API de Java que permite al programador ejecutar instrucciones en lenguaje estándar de acceso a bases de datos, **SQL** (*Structured Query Language*, lenguaje estructurado de consultas), que es un lenguaje de muy alto nivel que permite crear, examinar, manipular y gestionar bases de datos relacionales. Para que una aplicación pueda hacer operaciones en una base de datos, ha de tener una conexión con ella, que se establece a través de un *driver*, que convierte el lenguaje de alto nivel a sentencias de base de datos. Es decir, las tres acciones principales que realizará la JDBC son establecer la conexión a una base de datos, ya sea remota o no; enviar sentencias SQL a esa base de datos y; en tercer lugar, procesar los resultados obtenidos de la base de datos.

Una base de datos es una serie de tablas que contiene información ordenada en alguna estructura que facilita el acceso a esas tablas, el orden y la selección filas de las tablas según criterios específicos. Las bases de datos generalmente tienen *índices* asociados a alguna de sus columnas, de forma que el acceso sea lo más rápido posible.

El lenguaje en que se ha escrito la consulta es SQL, actualmente soportado por casi todas las bases de datos a lo largo del mundo. Los estándares de SQL han sido varios a lo largo de los años y muchas de las bases de datos para PC soportan alguno de esos tipos. El estándar **SQL-92** es el que se considera origen de todas las actualizaciones. Hay que tener en cuenta que hay posteriores versiones de SQL, perfeccionadas y extendidas para explotar características únicas de bases de datos particulares; así que no conviene separarse del estándar básico si se pretende hacer una aplicación que pueda interactuar con cualquier tipo de base de datos.

Desde que las PC se han convertido en una herramienta presente en la mayor parte de las oficinas, se han desarrollado un gran número de bases de datos para ejecutarse en ese tipo de plataformas; desde bases de datos muy elementales como *Microsoft Works*, hasta otras ya bastante sofisticadas como *Approach*, *dBase*, *Paradox*, *Access* y *Foxbase*. Otra categoría ya más seria de bases de datos para PC son aquellas que usan la plataforma PC como cliente para acceder a un servidor. Estas bases de datos son *IBM DB/2*, *Microsoft SQL Server*, *Oracle*, *Sybase*, *SQLBase*, *Informix*, *XDB* y *Postgres*. Todas estas bases de datos soportan varios dialectos similares de SQL, y todas parecen, a primera vista, intercambiables. La razón de que no sean intercambiables, por supuesto, es que cada una está diseñada con unas características de rendimiento distintas, con una interfaz de usuario y programación diferente. Aunque todas ellas soportan SQL y la programación es similar, cada base de datos tiene su propia forma de recibir las consultas SQL y su propio modo de devolver los resultados. Aquí es donde aparece el siguiente nivel de estandarización, de la mano de **ODBC** (*Open DataBase Conectivity*).

La idea es que se pueda escribir código independientemente de quien sea el propietario de la base de datos a la que se quiera acceder, de forma que se puedan extraer resultados similares de diferentes tipos de bases de datos sin necesidad de tocar el código del programa. Si se consiguiese escribir alguna forma de trazadores para estas

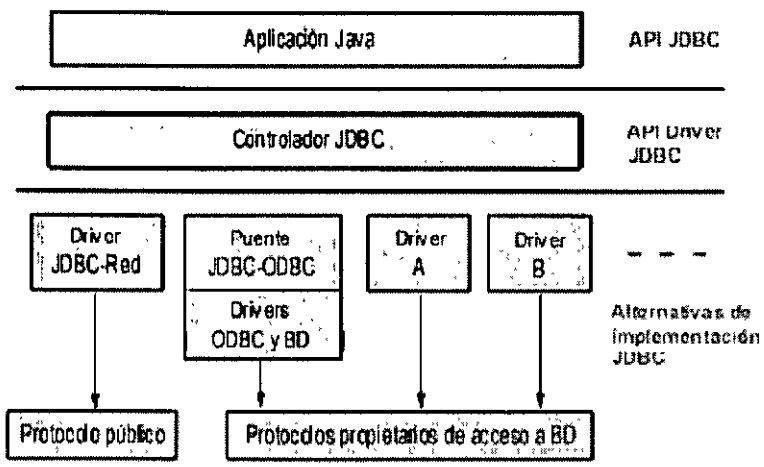
bases de datos que tuviesen una interfaz similar, el objetivo no sería difícil de alcanzar.

Sin embargo, ODBC dista mucho de ser la panacea que en un principio se podía pensar y que Microsoft se encargó de hacer creer. Muchos fabricantes de bases de datos soportan ODBC como una *interfaz alternativa* a la suya estándar, y la programación en ODBC no es trivial en absoluto; incluyendo toda la parafernalia de la programación para Windows con *handles*, apuntadores y opciones que son difíciles de asimilar. Finalmente, ODBC no es un estándar libre, ha sido desarrollado y es propiedad de Microsoft, lo cual, dados los vientos que soplan en este competitivo mundo de las compañías de software, hace su futuro difícil de predecir.

Para la gente del mundo Windows, JDBC es para Java lo que **ODBC** es para Windows. Windows en general no sabe acerca de las bases de datos, pero define el estándar ODBC consistente en un conjunto de primitivas que cualquier *driver* o fuente ODBC debe ser capaz de entender y manipular. Los programadores que a su vez deseen escribir programas para manejar bases de datos genéricas en Windows utilizan las funciones ODBC.

Con JDBC ocurre exactamente lo mismo. JDBC es una especificación de un conjunto de clases y métodos de operación que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea. Lógicamente, al igual que ODBC, la aplicación de Java debe tener acceso a un driver JDBC adecuado. Este driver es el que implementa la funcionalidad de todas las clases de acceso a datos y proporciona la comunicación entre el API JDBC y la base de datos real.

La necesidad de JDBC, a pesar de la existencia de ODBC, viene dada porque ODBC es una interfaz escrita en lenguaje C que, al no ser un lenguaje portable, hace que las aplicaciones Java también pierdan la portabilidad. Además, ODBC tiene el inconveniente de que se ha de instalar manualmente en cada máquina; al contrario que los drivers JDBC, que al estar escritos en Java son automáticamente instalables, portables y seguros.



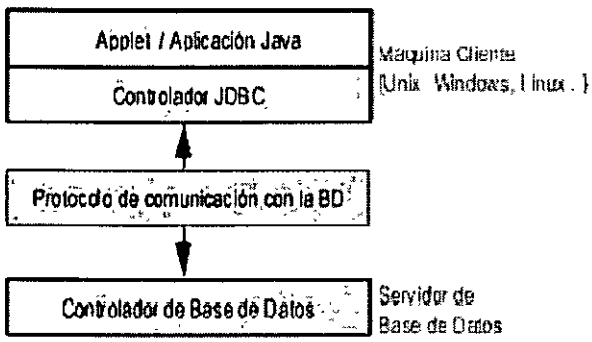
Toda la conectividad de bases de datos de Java se basa en sentencias SQL, por lo que se hace imprescindible un conocimiento adecuado de SQL para realizar cualquier clase de operación de bases de datos. Aunque, afortunadamente, casi todos los entornos de desarrollo Java ofrecen componentes visuales que proporcionan una funcionalidad suficientemente potente sin necesidad de que sea necesario utilizar SQL, aunque para usar directamente el JDK se haga imprescindible. La especificación JDBC requiere que cualquier driver JDBC sea compatible con al menos el nivel «de entrada» de ANSI SQL 92 (*ANSI SQL 92 Entry Level*).

2. Acceso de JDBC a bases de datos

El API JDBC soporta dos modelos diferentes de acceso a bases de datos, los modelos de dos y tres capas.

2.1 Modelo de dos capas

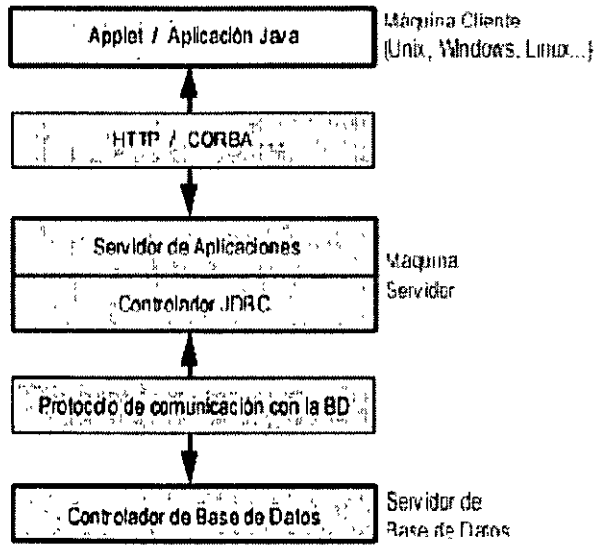
Este modelo se basa en que la conexión entre la aplicación Java o el applet que se ejecuta en el navegador, se conectan directamente a la base de datos.



Esto significa que el driver JDBC específico para conectarse con la base de datos debe residir en el sistema local. La base de datos puede estar en cualquier otra máquina y se accede a ella mediante la red. Esta es la configuración de típica *Cliente/Servidor*: el programa cliente envía instrucciones SQL a la base de datos, ésta las procesa y envía los resultados de vuelta a la aplicación.

2.2 Modelo de tres capas

En este modelo de acceso a las bases de datos, las instrucciones son enviadas a una capa intermedia entre Cliente y Servidor, que es la que se encarga de enviar las sentencias SQL a la base de datos y recoger el resultado desde la base de datos. En este caso, el usuario no tiene contacto directo, ni a través de la red, con la máquina donde reside la base de datos.



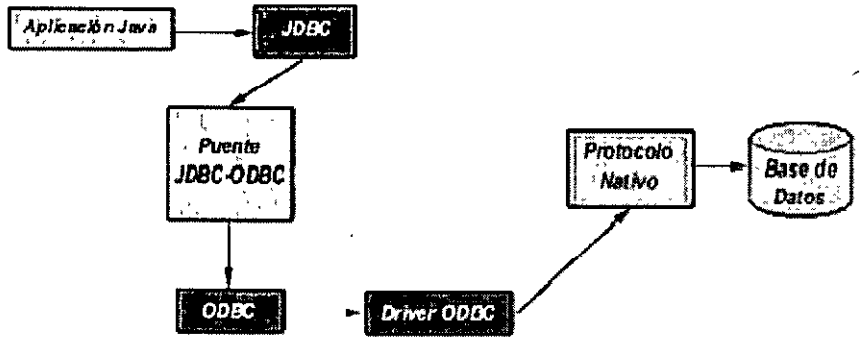
Este modelo presenta la ventaja de que el nivel intermedio mantiene en todo momento el control del tipo de operaciones que se realizan contra la base de datos, y además, está la ventaja adicional de que los drivers JDBC no tienen que residir en la máquina cliente, lo cual libera al usuario de la instalación de cualquier tipo de driver.

3. Tipos de drivers

Un driver JDBC puede pertenecer a una de cuatro categorías diferentes en cuanto a la forma de operar.

3.1 Puente JDBC-ODBC

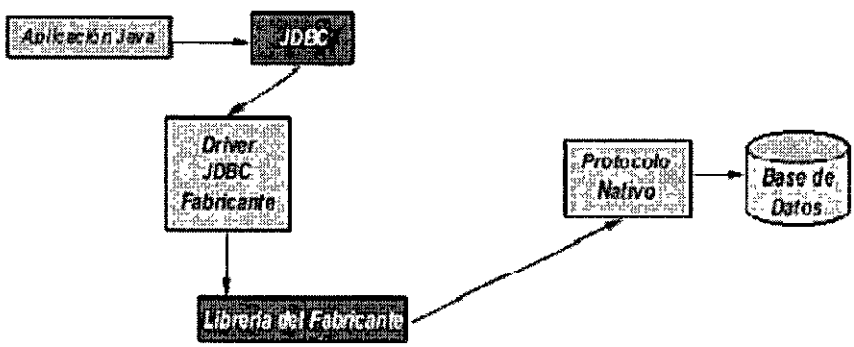
La primera categoría de drivers es la utilizada por Sun inicialmente para popularizar JDBC y consiste en aprovechar todo lo existente, estableciendo un puente entre JDBC y ODBC. Este driver convierte todas las llamadas JDBC a llamadas ODBC y realiza la conversión correspondiente de los resultados.



La ventaja de este driver que se proporciona con el JDK, es que Java dispone de acceso inmediato a todas las fuentes posibles de bases de datos y no hay que hacer ninguna configuración adicional además de la ya existente. No obstante, tiene dos desventajas muy importantes: por un lado, la mayoría de los drivers ODBC a su vez convierten sus llamadas a llamadas de una librería nativa del fabricante DBMS, con lo cual la lentitud del driver JDBC-ODBC puede ser exasperante, al llevar dos capas adicionales que no añaden funcionalidad alguna; y por otra parte, el puente JDBC-ODBC requiere una instalación ODBC ya existente y configurada.

Lo anterior implica que para distribuir con seguridad una aplicación Java que use JDBC habría que limitarse en primer lugar a entornos Windows (donde está definido ODBC) y en segundo lugar, proporcionar los drivers ODBC adecuados y configurarlos correctamente. Esto hace que este tipo de drivers esté totalmente descartado en el caso de aplicaciones comerciales, e incluso en cualquier otro desarrollo, debe ser considerado como una solución transitoria, porque el desarrollo de drivers totalmente en Java hará innecesario el uso de estos puentes.

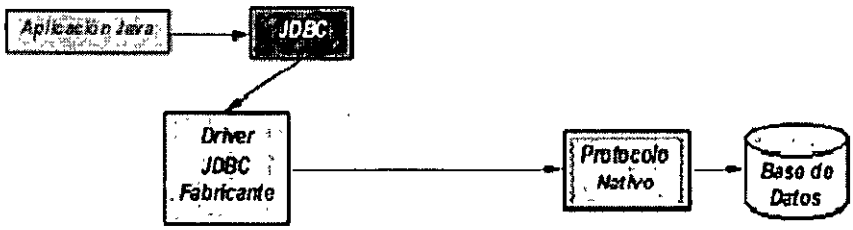
3.2 Java/Binario



Este *driver* se salta la capa ODBC y habla directamente con la librería nativa del fabricante del sistema DBMS (como pudiera ser *DB-Library* para Microsoft SQL Server o *CT-Lib* para Sybase SQL Server). Este driver es uno 100% Java, pero aún así necesita la existencia de un código binario (la librería DBMS) en la máquina del cliente, con las limitaciones y problemas que esto implica.

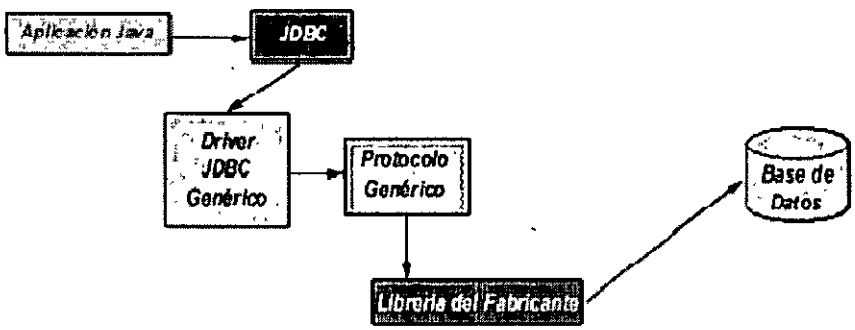
3.3 100% Java/Protocolo nativo

Es un driver realizado completamente en Java que se comunica con el servidor DBMS utilizando el protocolo de red nativo del servidor. De esta forma, el driver no necesita intermediarios para hablar con el servidor y convierte todas las peticiones JDBC en peticiones de red contra el servidor. La ventaja de este tipo de driver es que es una solución 100% Java y, por lo tanto, independiente de la máquina en la que se va a ejecutar el programa.



Igualmente, dependiendo de la forma en que esté programado el driver, puede no necesitar ninguna clase de configuración por parte del usuario. La única desventaja de este tipo de *drivers* es que el cliente está ligado a un servidor DBMS concreto, ya que el protocolo de red que utiliza MS SQL Server por ejemplo no tiene nada que ver con el utilizado por DB2, PostGres u Oracle. La mayoría de los fabricantes de bases de datos han incorporado a sus propios drivers JDBC del segundo o tercer tipo, con la ventaja de que no suponen un coste adicional.

3.4 100% Java/Protocolo independiente

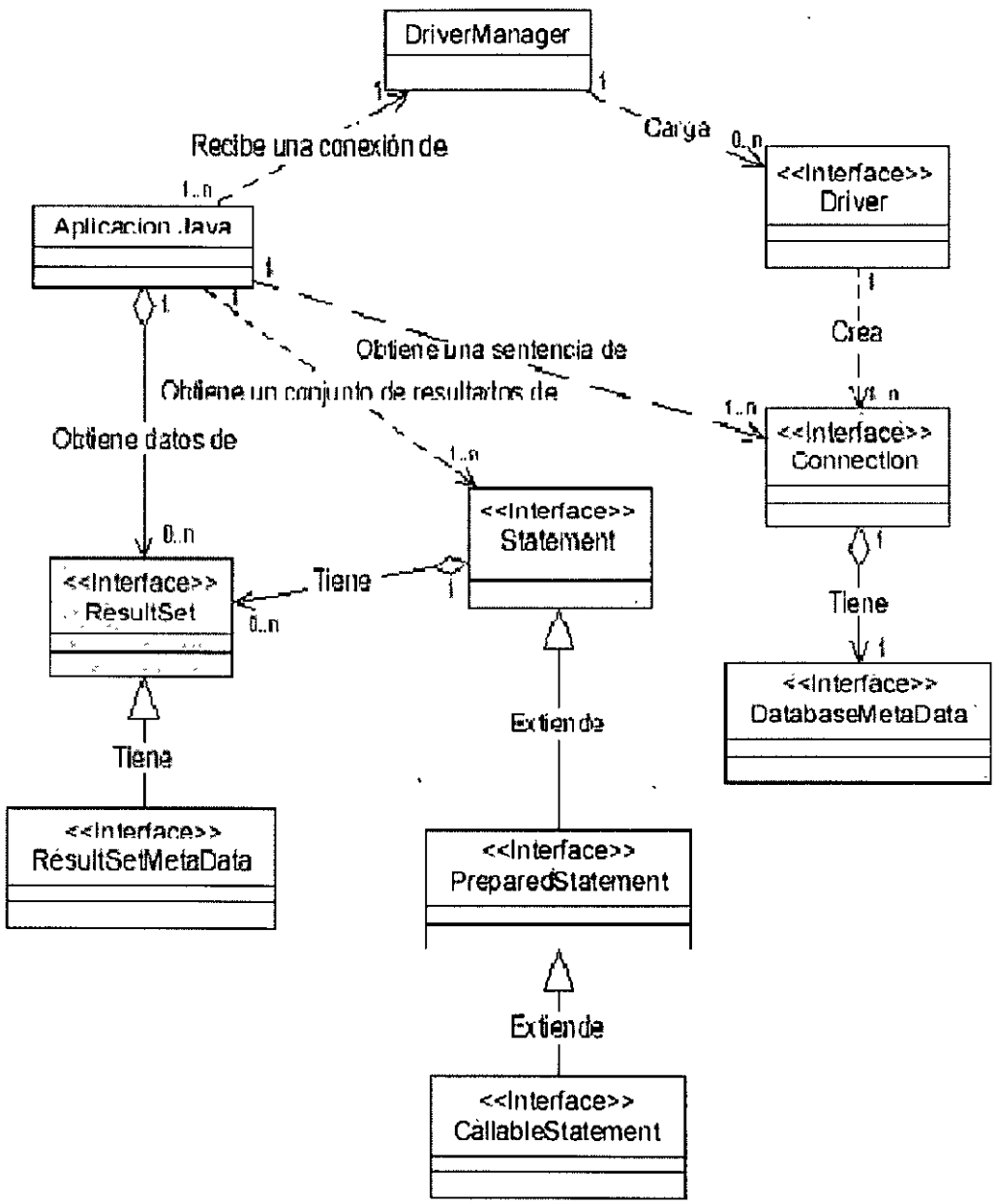


Esta es la opción más flexible, se trata de un driver 100% Java / Protocolo independiente, que requiere la presencia de un intermediario en el servidor. En este caso, el driver JDBC hace las peticiones de datos al intermediario en un protocolo de red independiente del servidor DBMS. El intermediario a su vez, que está ubicado en el lado del servidor, convierte las peticiones JDBC en peticiones nativas del sistema DBMS. La ventaja de este método es inmediata: el programa que se ejecuta en el cliente, y aparte de las ventajas de los drivers 100% Java, también presenta la independencia respecto al sistema de bases de datos que se encuentra en el servidor.

De esta forma, si una empresa distribuye una aplicación Java para que sus usuarios puedan acceder a su servidor MS SQL y posteriormente decide cambiar el servidor por Oracle, PostGres o DB2, no necesita volver a distribuir la aplicación, sino que únicamente debe reconfigurar la aplicación residente en el servidor que se encarga de transformar las peticiones de red en peticiones nativas. La única desventaja de este tipo de drivers es que la aplicación intermediaria es una aplicación independiente que suele tener un coste adicional por servidor físico, que hay que añadir al coste del servidor de bases de datos.

4. UML de la JDBC

JDBC define ocho interfaces para operaciones con bases de datos, de las que se derivan las clases correspondientes. La figura siguiente, en formato OMT, con nomenclatura UML, muestra la interrelación entre estas clases según el modelo de objetos de la especificación de JDBC.



5. Proceso de Conexión

La clase que se encarga de cargar inicialmente todos los drivers JDBC disponibles es **DriverManager**. Una aplicación puede utilizar **DriverManager** para obtener un objeto de tipo conexión, **Connection**, con una base de datos. La conexión se especifica siguiendo una sintaxis basada en la especificación más amplia de los URL, de la forma

jdbc:subprotocolo//servidor:puerto/base de datos

Por ejemplo, si se utiliza mSQL el nombre del subprotocolo será **msql**. En algunas ocasiones es necesario identificar aún más el protocolo. Por ejemplo, si se usa el puente JDBC-ODBC no es suficiente con **jdbc:odbc**, ya que pueden existir múltiples drivers ODBC, y en este caso, hay que especificar aún más, mediante **jdbc:odbc:fuente de datos**.

Una vez que se tiene un objeto de tipo **Connection**, se pueden crear sentencias, **statements**, ejecutables. Cada una de estas sentencias puede devolver cero o más resultados, que se devuelven como objetos de tipo **ResultSet**. La tabla siguiente muestra la misma lista de clases e interfaces junto con una breve descripción.

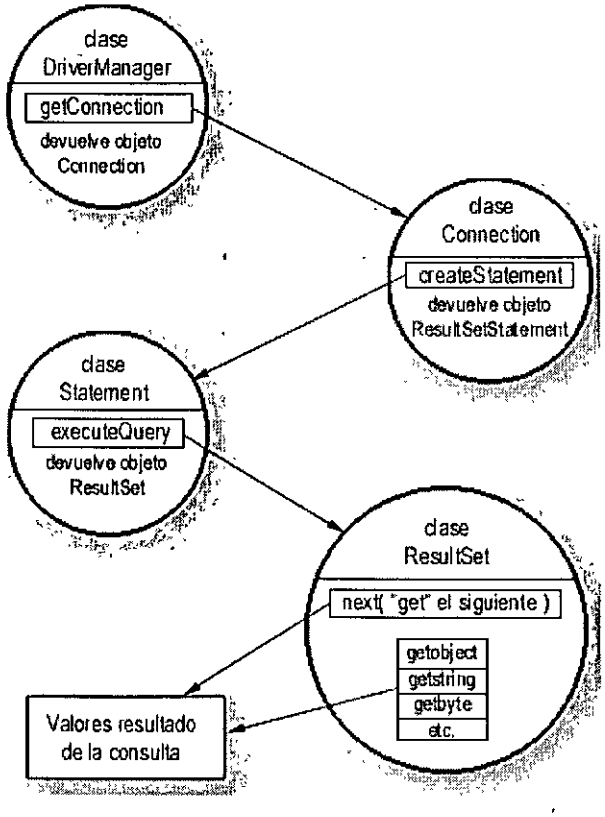
Clase/Interface	*Descripción*
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto
DriverManager	Permite gestionar todos los drivers instalados en el sistema
DriverPropertyInfo	Proporciona diversa información acerca de un driver
Connection	Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión a más de una base de datos
DatabaseMetadata	Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc.
Statement	Permite ejecutar sentencias SQL sin parámetros
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada/TD>
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, típicamente procedimientos almacenados
ResultSet	Contiene las filas o registros obtenidos al ejecutar un

SELECT

ResultSetMetadata	Permite obtener información sobre un **ResultSet**, como el número de columnas, sus nombres, etc.

Si se desea utilizar otra fuente ODBC, no tiene más que cambiarse los parámetros de *getConnection()* en el código fuente. El establecimiento de la conexión es fácil, como se puede suponer. Si algo no funciona, cosa más que probable en los primeros intentos, es muy recomendable activar la traza de llamadas ODBC desde el panel de control. De esta forma se puede ver lo que está haciendo exactamente el *driver* JDBC y por qué motivo no se está estableciendo la conexión.

El siguiente diagrama relaciona las cuatro clases principales que va a usar cualquier programa Java con JDBC, y representa el esqueleto de cualquiera de los programas que se desarrollan para atacar a bases de datos.



Lo primero que se hace es importar toda la funcionalidad de JDBC, a través de la primera sentencia ejecutable del programa.

```
import java.sql.*;
```

Las siguientes líneas son las que cargan el puente JDBC-ODBC, mediante el método *forName()* de la clase **Class**.

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
```

En teoría esto no es necesario, ya que **DriverManager** se encarga de leer todos los *drivers* JDBC compatibles, pero no siempre ocurre así, por lo que es mejor asegurarse. El método *forName()* localiza, lee y enlaza dinámicamente una clase determinada. Para *drivers* JDBC, la sintaxis que se recomienda de *forName()* es *nombreEmpresa.nombreBaseDatos.nombreDriver*, y el *driver* deberá estar ubicado en el directorio *nombreEmpresa\nombreBaseDatos\nombreDriver.class* a partir del directorio indicado por la variable de entorno *CLASSPATH*. En este caso se indica que el puente JDBC-ODBC que se desea leer es precisamente el de Sun.

Si por cualquier motivo no es posible conseguir cargar **JdbcOdbcDriver.class**, se intercepta la excepción y se sale del programa. En este momento es la hora de echar mano de la información que puedan proporcionar las trazas ODBC.

La carga del driver también se puede especificar desde la línea de comandos al lanzar la aplicación:

```
java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver ElPrograma
```

A continuación, se solicita a **DriverManager** que proporcione una conexión para una fuente de datos ODBC. El parámetro **jdbc:odbc:Tutorial** especifica que la intención es acceder a la fuente de datos con nombre **Tutorial**, *Data Source Name* o DSN, en la terminología ODBC.

```
conexion = DriverManager.getConnection("jdbc:odbc:Tutorial","","" );
```

El segundo y tercer parámetro son el nombre del usuario y la clave con la cual se intentará la conexión. En este caso el acceso es libre, para acceder como administrador del sistema en el caso de un servidor MS SQL se usa la cuenta *sa* o *system administrator*, cuya cuenta de acceso no tiene clave definida; en caso de acceder a un servidor MS Access, la cuenta del administrador es *admin* y también sin clave definida. Esta es la única línea que con seguridad habrá de cambiar el programador para probar sus aplicaciones. *getConnection* admite también una forma con un único parámetro (el URL de la base de datos), que debe proporcionar toda la información de conexión necesaria al *driver* JDBC correspondiente. Para el caso JDBC-ODBC, se puede utilizar la sentencia equivalente:

```
DriverManager.getConnection ( "jdbc:odbc:SQL;UID=sa;PWD=" );
```

Para el resto de los *drivers* JDBC, habrá que consultar la documentación de cada *driver* en concreto. Inmediatamente después de obtener la conexión, en la siguiente línea:

sentencia = conexion.createStatement();

se solicita que proporcione un objeto de tipo *Statement* para poder ejecutar sentencias a través de esa conexión. Para ello se dispone de los métodos *execute(String sentencia)* para ejecutar una petición SQL que no devuelve datos o *executeQuery(String sentencia)* para ejecutar una consulta SQL. Este último método devuelve un objeto de tipo *ResultSet*.

Una vez que se tiene el objeto *Statement* ya se pueden lanzar consultas y ejecutar sentencias contra el servidor. A partir de aquí el resto del programa realmente es SQL «adornado».

Usuando el método *next()* la posición se situará en el «siguiente» elemento del resultado, o bien sobre el primero si todavía no se ha utilizado. La función *next()* devuelve *true* o *false* si el elemento existe, de forma que se puede iterar mediante *while (resultado.next())* para tener acceso a todos los elementos.

A continuación, en las líneas siguientes se utilizan los métodos *getXXX()* de resultado para tener acceso a las diferentes columnas. El acceso se puede hacer por el nombre de la columna, como en las dos primeras líneas, o bien mediante su ubicación relativa, como en la última línea. Además de *getString()* están disponibles *getBoolean()*, *getByte()*, *getDouble()*, *getFloat()*, *getInt()*, *getLong()*, *getNumeric()*, *getObject()*, *getShort()*, *getDate()*, *getTime()* y *getUnicodeStream()*, cada uno de los cuales devuelve la columna en el formato correspondiente, si es posible.

Después de haber trabajado con una sentencia o una conexión es recomendable cerrarla mediante *sentencia.close()* o *conexión.close()*. De forma predeterminada los *drivers* JDBC deben hacer un *COMMIT* de cada sentencia. Este comportamiento se puede modificar mediante el método *Connection.setAutoCommit(boolean nuevovalor)*. En el caso de que se establezca *AutoCommit* a *false*, será necesario llamar de forma explícita a *Connection.commit()* para guardar los cambios realizados o *Connection.rollback()* para deshacerlos.

Como se habrá podido comprobar hasta ahora, no hay nada intrínsecamente difícil en conectar Java con una base de datos remota. Los posibles problemas de conexión que puede haber (selección del *driver* o fuente de datos adecuada, obtención de acceso, etc.), son problemas que se tendrían de una u otra forma en cualquier lenguaje de programación.

El objeto **ResultSet** devuelto por el método *executeQuery()*, permite recorrer las filas obtenidas, no proporciona información referente a la estructura de cada una de ellas; para ello se utiliza **ResultSetMetaData**, que permite obtener el tipo de cada campo o columna, su nombre, si es del tipo autoincremento, si es sensible a mayúsculas, si se puede escribir en dicha columna, si admite valores nulos, etc.

Para obtener un objeto de tipo **ResultSetMetaData** basta con llamar al método *getMetaData()* del objeto **ResultSet**. En la lista siguiente aparecen algunos de los métodos más importantes de **ResultSetMetaData**, que permiten averiguar toda la información necesaria para formatear la información correspondiente a una columna, etc.

getCatalogName()
Nombre de la columna en el catálogo de la base de datos

getColumnName()
Nombre de la columna

getColumnLabel()
Nombre a utilizar a la hora de imprimir el nombre de la columna

getColumnDisplaySize()
Ancho máximo en caracteres necesario para mostrar el contenido de la columna

getColumnCount()
Número de columnas en el **ResultSet**

getTableName()
Nombre de la tabla a que pertenece la columna

getPrecision()
Número de dígitos de la columna

getScale()
Número de decimales para la columna

getColumnType()
Tipo de la columna (uno de los tipos SQL en **java.sql.Types**)

getColumnTypeName()
Nombre del tipo de la columna

isSigned()
Para números, indica si la columna corresponde a un número con signo

isAutoIncrement()
Indica si la columna es de tipo autoincremento

isCurrency()
Indica si la columna contiene un valor monetario

isCaseSensitive()
Indica si la columna contiene un texto sensible a mayúsculas

isNullable()
Indica si la columna puede contener un NULL SQL. Puede devolver los valores columnNoNulls, columnNullable o columnNullableUnknown, miembros finales estáticos de **ResultSetMetaData** (constantes)

isReadOnly()
Indica si la columna es de solo lectura

isWritable()
Indica si la columna puede modificarse, aunque no lo garantiza

isDefinitivelyWritable()
Indica si es absolutamente seguro que la columna se puede modificar

isSearchable()
Indica si es posible utilizar la columna para determinar los criterios de búsqueda de un SELECT

getSchemaName()
Devuelve el texto correspondiente al esquema de la base de datos para esa columna

6. Objetos en la conexión a base de datos

En general pues, los objetos que se van a poder encontrar en una aplicación que utilice JDBC, serán los que se indican a continuación.

Connection

Representa la conexión con la base de datos. Es el objeto que permite realizar las consultas SQL y obtener los resultados de dichas consultas. Es el objeto base para la creación de los objetos de acceso a la base de datos.

Métodos de la interfaz Connection

Tipo de valor de retorno del método	Método
void	**clearWarnings** () Borra todos los avisos de los que se ha informado para este objeto Connection.
void	**close** () Libera una base de datos de Connection y los recursos de JDBC de modo inmediato en vez de esperar que los mismos se liberen de modo automático.
void	**commit** () Hace que sean permanentes los cambios hechos desde la última operación de confirmación o cancelación y libera los bloqueos de base de datos mantenidos actualmente por

	Connection.
Statement	**createStatement**() Crea un objeto Statement para enviar sentencias de SQL a la base de datos.
Statement	**createStatement**(int resultSetType, int resultSetConcurrency) JDBC 2.0. Crea un objeto Statement que generará objetos ResultSet con el tipo y la simultaneidad indicados.
boolean	**isClosed**() Comprueba si se ha cerrado una Conexión.
DatabaseMetaData	**getMetaData**() Obtiene los metadatos relativos a esta base de datos de Connection.
SQLWarning	**getWarnings**() Devuelve el primer aviso informado por llamadas a esta Connection.
CallableStatement	**prepareCall**(String sql) Crea un objeto CallableStatement para llamar procedimientos almacenados de base de datos.
PreparedStatement	**prepareStatement** (String sql) Crea un objeto PreparedStatement para enviar sentencias de SQL parametrizadas a la base de datos.
PreparedStatement	**prepareStatement**(String sql, int resultSetType, int resultSetConcurrency) JDBC 2.0. Crea un objeto PreparedStatement que generará objetos ResultSet con el tipo y la simultaneidad indicados.
void	**rollback** () Elimina los cambios hechos desde la última operación de confirmación o cancelación y libera los bloqueos de base de datos mantenidos actualmente por esta Connection.
void	**setAutoCommit** (boolean autoCommit) Establece la modalidad de confirmación automática de esta conexión.

DriverManager

Encargado de mantener los drivers que están disponibles en una aplicación concreta. Es el objeto que mantiene las funciones de administración de las operaciones que se realizan con la base de datos.

Statement

Se utiliza para enviar las sentencias SQL simples, aquellas que no necesitan parámetros, a la base de datos.

PreparedStatement

Tiene una relación de herencia con el objeto **Statement**, añadiéndole la funcionalidad de poder utilizar parámetros de entrada. Además, tiene la particularidad de que la pregunta ya ha sido compilada antes de ser realizada, por loque se denomina *preparada* La principal ventaja, aparte de la utilización de parámetros, es la rapidez de ejecución de la pregunta.

```
ps=con.prepareStatement("insert into registration (
                        theuser, password,
                        emailaddress, creditcard,
                        balance) values (
                        ?, ?, ?, ?, ?)");
  ps.setString(1, theuser);
  ps.setString(2, password);
  ps.setString(3, emailaddress);
  ps.setString(4, creditcard);
  ps.setDouble(5, balance);
```

CallableStatement

Tiene una relación de herencia cn el objeto PreparedStatement. Permite utilizar funciones implementadas directamente sobre el sistema de gestión de la base de datos. Teniendo en cuenta que éste posee información adicional sobre el uso de las estructuras internas, índices, etc.; las funciones se realizarán de forma más eficiente. Este tipo de operaciones es muy utilizada en el caso de ser funciones muy complicadas o bien que vayan a ser ejecutadas varias veces a lo largo del tiempo de vida de la aplicación.

ResultSet

Contiene la tabla resultado de la pregunta SQL que se haya realizado. En párrafos anteriores se han comentado los métodos que proporciona este objeto para recorrer dicha tabla.

7. Información de la Base de Datos

Falta aún una pieza importante a la hora de trabajar con la conexión a la base de datos mediante **Connection**, y es la posibilidad de poder interrogar sobre las características de una base de datos; por ejemplo, puede se interesante saber si la base de datos soporta cierto nivel de aislamiento en una transacción, como la TRANSACTION_SERIALIZABLE, que muchos gestores no soportan. Para esto está otro de los interfaces que proporciona JDBC, **DatabaseMetaData**, al que es posible interrogar sobre las características de la base de datos con la que se está trabajando. Es posible obtener un objeto de tipo **DatabaseMetaData** mediante el método *getMetaData()* de **Connection**.

DatabaseMetaData proporciona diversa información sobre una base de datos, y cuenta con varias docenas de métodos, a través de los cuales es posible obtener gran cantidad de información acerca de una tabla; por ejemplo, *getColumns()* devuelve las columnas de una tabla, *getPrimaryKeys()* devuelve la lista de columnas que forman la clave primaria, *getIndexInfo()* devuelve información acerca de sus índices, mientras que *getExportedKeys()* devuelve la lista de todas las claves ajenas que utilizan la clave primaria de esta tabla, y *getImportedKeys()* las claves ajenas existentes en la tabla. El método *getTables()* devuelve la lista de todas las tablas en la base de datos, mientras que *getProcedures()* devuelve la lista de procedimientos almacenados. Muchos de los métodos de **DatabaseMetaData** devuelven un objeto de tipo **ResultSet** que contiene la información deseada. El listado que se presenta a continuación, muestra el código necesario para obtener todas las tablas de una base de datos.

Hay todo un grupo de métodos que permiten averiguar si ciertas características están soportadas por la base de datos; entre ellos, destacan *supportsGroupBy()* indica si se soporta el uso de GROUP BY en un SELECT, mientras que *supportsOuterJoins()* indica si se pueden llevar a cabo *outer-joins*. El método *supportsTransactions()*, comentado antes, indica si cierto tipo de transacciones está soportado o no. Otros métodos de utilidad son *getUserName()*, que devuelve el nombre del usuario actual; *getURL()*, que devuelve el URL de la base de datos actual.

DatabaseMetaData proporciona muchos otros métodos que permiten averiguar cosas tales como el máximo número de columnas utilizable en un SELECT, etc. En general, casi cualquier pregunta sobre las capacidades de la base de datos se puede contestar llamando a los distintos métodos del objeto **DatabaseMetaData**, que merece la pena que el lector consulte cuando no sepa si cierta característica está soportada.

8. Tipos SQL en Java

Muchos de los tipos de datos estándar de SQL '92, no tienen un equivalente nativo en Java. Para superar esta deficiencia, se deben mapear los tipos de datos SQL en Java, utilizando las clases JDBC para acceder a los tipos de datos SQL. Es necesario saber cómo recuperar adecuadamente tipos de datos Java; como *int*, *long*, o *string*, a partir de sus contrapartidas SQL almacenadas en base de datos. Esto puede ser especialmente importante si se está trabajando con datos numéricos, que necesiten control decimal con precisión, o con fechas SQL, que tienen un formato muy bien definido.

El mapeo de los tipos de datos Java a SQL es realmente sencillo, tal como se muestra en la tabla que acompaña a este párrafo. Observe el lector que los tipos que comienzan por "java" no son tipos básicos, sino clases que tienen métodos para trasladar los datos a formatos utilizables, y son necesarias estas clases porque no hay un tipo de datos básico que mapee directamente su contrapartida SQL. La creación de estas clases debe hacerse siempre que se necesite almacenar un tipo de dato SQL en un programa Java, para poder utilizar directamente el dato desde la base de datos.

Java	SQL

String	VARCHAR
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]-byte array: imagenes, sonidos...	VARBINARY (BLOBs)
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.math.BigDecimal	NUMERIC

El tipo de dato *byte[]*, es un array de bytes de tamaño variable. Esta estructura de datos guarda datos binarios, que en SQL son VARBINARY y LONG-VARBINARY. Estos tipos se utilizan para almacenar imágenes, ficheros de documentos, y cosas parecidas. Para almacenar y recuperar este tipo de información de la base de datos, se deben utilizar los métodos para streams que proporciona JDBC: *setBinaryStream()* y *getBinaryStream()*.

La conversión de tipos en el sentido contrario puede no estar tan clara, ya que hay tipos SQL cuya tipo Java correspondiente puede no ser evidente, como VARBINARY, o DECIMAL, etc. La tabla siguiente muestra los tipos Java correspondientes a cada tipo SQL.

SQL	**Java**
CHAR	String
VARCHAR	String
LONGVARCHAR	String

NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Existe una constante para cada tipo de dato SQL, declarada en **java.sql.Types**; por ejemplo, el tipo al tipo TIMESTAMP le corresponde la constante **java.sql.Types.TIMESTAMP**.

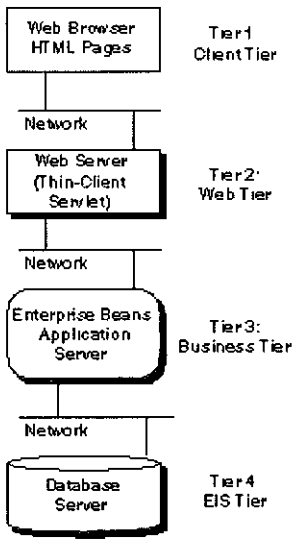
Además, JDBC proporciona clases Java nuevas para representar varios tipos de datos SQL: estas son **java.sql.Date**, **java.sql.Time** y **java.sql.Timestamp**.

II. J2EE

1. Tecnología J2EE

1.1 Aplicación Cliente Multi-Capa

Las aplicaciones multi-capa pueden consistir en 3 ó 4 capas.



1.2 Software J2EE y Configuración

Para ejecutar los ejemplos del tutorial, necesitas descargar e instalar el SDK Java 2 Enterprise Edition (J2EE), Versión 1.2.1 (http://java.sun.com/j2ee/download.html), y SDK Java 2, Standard Edition (J2SE), Versión 1.2 o posterior (http://java.sun.com/jdk/index.html). Las instrucciones de este tutorial asumen que J2EE y J2SE están instalados en el directorio J2EE debajo del directorio home del usuario.

Unix:

/home/monicap/J2EE/j2sdkee1.2.1

/home/monicap/J2EE/jdk1.2.2

Windows:

\home\monicap\J2EE\j2sdkee1.2.1

\home\monicap\J2EE\jdk1.2.2

1.3 Configuración del Path y ClassPath

La descarga contiene el servidor de aplicaciones J2EE, la base de datos Cloudscape, un servidor Web que usa capas de socket seguras (SSL) también conocido como HTTP sobre HTTPS, herramientas de desarrollo y despliegue, y los APIs Java para Enterprise. Para usar estar características, debemos configurar las variables de entorno path y classpath como se describe aquí:

1.3.1 Configuración del Path

La configuración del Path hace accesibles las herramientas de desarrollo y despliegue desde cualquier lugar de nuestro sistema. Debemos asegurarnos de seleccionar estos path antes de cualquier otro path que pudiéramos tener de viejas instalaciones del JDK.

Unix:

/home/monicap/J2EE/jdk1.2.2/bin

/home/monicap/J2EE/j2sdkee1.2.1/bin

Windows:

\home\monicap\J2EE\jdk1.2.2\bin

\home\monicap\J2EE\j2sdkee1.2.1\bin

1.3.2 Configuración del ClassPath

La configuración del.ClassPath le dice a las herramientas de desarrollo y despliegue de Java 2 dónde encontrar las distintas librerías de clases que usa:

Unix:

/home/monicap/J2EE/j2sdkee1.2.1/lib/j2ee.jar

Windows:

\home\monicap\J2EE\j2sdkee1.2.1\lib\j2ee.jar

1.4 Componentes de Aplicación J2EE

Los programadores de aplicaciones J2EE escriben componentes de aplicación J2EE. Un componente J2EE es una unidad de software funcional auto-contenida que se ensambla dentro de una aplicación J2EE y que se comunica con otros componentes de aplicación. La especificación J2EE define los siguientes componentes de aplicación:

- Componentes de Aplicación Cliente
- Componentes JavaBeans
- Componentes Servlets y JavaServer Pages (también llamados componentes Web)
- Applets

2. Servlets

2.1 Introducción a los Servlets

Los servlets son módulos java que sirven para extender las capacidades de los servidores Web. Aunque es una definición un poco ambigua los servlets son programas para los servidores, mientras que los applets son programas para los clientes. Dentro de una evolución cronológica los servlets son la siguiente etapa de los CGI. En algunas bibliografías son referenciados como CGI de 2ª generación, la cual comparten con lenguajes como ASP, PHP y JSP (que al fin y al cabo son servlets). Aparecieron en 1997 y poco a poco se han convirtiendo en el entorno dominante de la programación Java en servidor.

El uso de los servlets permite elevar el del desarrollo de páginas Web dinámicas (en contenido y diseño), apoyándose además en la potencia que nos proporciona el lenguaje Java. Puede desarrollarse desde un simple servlet que muestre una página Web simple saludando, hasta uno que se conecte a una base de datos utilizando un pool de conexiones, encriptando la información en su envío, accediendo a bases de datos distribuidas y manteniendo su información de forma persistente en un EJB; todo ello para conseguir una información dinámica. A partir de aquí las posibilidades son infinitas.

Describir un servlet es como describir una máquina de estados. Desde el momento que se inicia el servlet hasta que el servlet es destruido, éste pasará por una serie de estados dependiendo de cada una de las situaciones ante las que se encuentre.

De modo general, puede afirmarse que la secuencia de acciones que se producen en un servlet son las siguientes. La primera vez que se realice una petición sobre el servlet se ejecutará un método de inicio, denominado init, en el cual se inician las variables necesarias. Una vez que realizado esto, el servlet se encuentra a la escucha en espera de peticiones. Cada una de las peticiones que se reciban será atendida en un hilo de ejecución diferente, a no ser que se indique lo contrario. Dependiendo de como lleguen los datos (mediante post o get) al servlet se ejecutará un método u otro, doPost o doGet. Por último el servlet tendrá un estado de finalización en el cual eliminará las variables creadas en su inicialización, conexiones a bases de datos. Este el método destroy.

2.2 Codificación

A la hora de codificar, lo primero que debe saberse es que el servlet deberá heredar de la clase HttpServlet la cual contendrá todos los métodos necesarios para generar un servlet. Dicha clase se encuentra en el paquete javax.servlet.

```
import javax.servlet.*;
public class MiServlet extends HttpServlet {}
```

Solamente debe sobrescribirse aquellos métodos que sea oportuno implementar en el servlet. Recuérdese además que un servlet es una clase de Java, como cualquiera otra, de manera que, de ser necesario, puede agregarse otros métodos o atributos

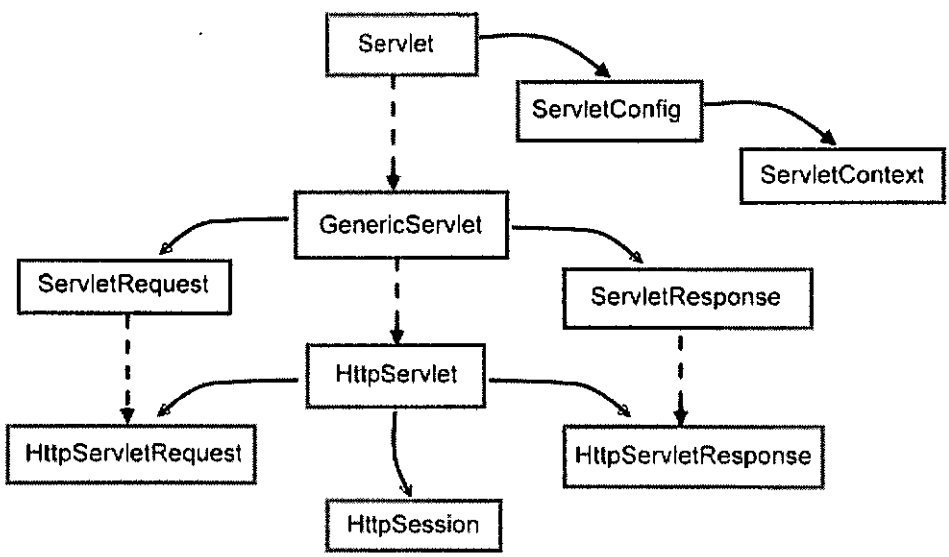
2.3 Características de los servlets

Dentro de las características que presenta la plataforma de desarrollo de servlets pueden enumerarse las siguientes:

1. Es independiente de la plataforma en la que se este ejecutando. Otras soluciones como ISAPI o NSAPI son dependientes de la plataforma y de los servidores donde se ejecuta haciendo muy costoso una migración en la plataforma de ejecución.
2. Ejecución multihilo. Cada una de las peticiones sobre el servlet creará una instancia que se ejecutará de manera independiente. A no ser de que le indiquemos lo contrario. El servlet permanece cargado en memoria por lo que atiende rápidamente las peticiones.
3. Un servlet puede ejecutarse en un sandbox, lo cual limita los privilegios del servlet a un modelo controlado como el de los applets, salvaguardando la integridad del servidor donde se ejecuta.
4. Un servlet puede llamar a otro servlet, incluso a métodos de otros servlets. Esto permite que un servlet realice balanceado de carga entre diferentes servlets. Además, desde un servlet, puede redirigirse una petición sobre otro servlet (en la misma máquina o en una máquina remota).
5. El servlet puede obtener información acerca de la maquina que ha realizado la petición (IP, puerto, tipo de método de envío: get o post,...).
6. Uno de los problemas del protocolo HTTP es que es un protocolo sin estado. No existe una relación entre las diferentes peticiones HTTP realizadas por un usuario sobre un servidor, sino que tiene que ser el propio servidor el que mantenga esta sesión. Para mantener algún tipo de información del usuario (su identificación, los productos comprados en las diferentes pantallas,...) en los servlets, se puede utilizar las sesiones y cookies para poder llevar a acabo esto. La única diferencia es que, en las sesiones, la información del usuario se almacena en el servidor, mientras que con las cookies la información del usuario se almacena en su propia máquina.

7. Conexión a Bases de Datos. A través de los servlets es posible establecer conexiones a diferentes tipos de bases de datos. Esta característica acopla perfectamente a los servlets dentro de una arquitectura cliente/servidor en 3 capas (cliente - servidor - datos).
8. Proxy para applets. Dentro del desarrollo de applets surgen gran número de limitaciones, dentro de las cuales el acceso al sistema de archivos es una principal. Para subsanar dicha carencia puede interponerse un servlet entre el applet y el sistema de archivos, de tal manera que el applet se comunique con el servlet, que será el encargado de acceder al sistema de archivos.
9. Generación dinámica de código. Esta es una de las características más utilizadas en los servlets, la generación dinámica de HTML. Esto permite que una misma página tenga múltiples salidas o representaciones en cuanto a estructura y contenido atendiendo a las evaluaciones que tome el servlet: ip del usuario, información de una base de datos, fecha del sistema, etc.
10. Recursos compartidos entre usuarios. Los servlets pueden definir estructuras o información que va a ser compartida por diferentes usuarios que utilicen el servlet. A la hora de utilizar esta información compartida o global deberemos de tomar las precauciones oportunas para que siempre sea una información correcta, integra y fiable.
11. Mediante el manejo de excepciones los errores pueden manejarse sencillamente durante la ejecución del servlet.
12. Son 100% puro Java, lo que los hacen multiplataforma.

2.4 Clases involucradas



2.5 Servlets vs. Applets

- Limitación al acceso de otro servidor que no sea el mismo en que se aloje el applet.
- Limitación en el acceso a los servicios del servidor
- Limitaciones de seguridad. (ingeniería inversa para retocar el applet, para acceso a la lógica del negocio del sistema en cuestión)

2.6 Ventajas de los Servlets vs CGI

Hoy en día las aplicaciones CGI tienden a desaparecer debido a los siguientes inconvenientes:

- Como están diseñados para lenguajes interpretados como el PERL disminuían el rendimiento de los programas que se ejecutaban. Una alternativa era compilarlos, pero no eran la mejor alternativa por la dependencia que se generaba con la plataforma en que se compilaban
- Como todo programa CGI se ejecutaba como un proceso independiente, se puede llegar a la conclusión, que el arranque de todo CGI requiere un tiempo bastante importante y que a al momento de realizar aplicaciones críticas con ellos se hace casi imposible.
- Cuando se desea que un CGI invoque a otro CGI se tiene el problema de ejecución de dos procesos independientes, además la comunicación entre ellos se debía hacer mediante archivos, lo que da una idea del tiempo que tomaba acceder a dichos archivos. Es por lo tanto una programación muy difícil si lo que se quería lograr eran tiempos razonables a costa de la sobrecarga del servidor.

2.7 Métodos

public abstract interface Servlet: Todos los servlets implementan este interfaz directamente o extendiendo una clase que lo implemente como HttpServlet.
Entre sus métodos están:

- **init(ServletConfig config)**: Es el método utilizado para crear una nueva instancia del servlet (análogo al constructor). Ver el ciclo de vida. Este método puede ser sobreescrito para realizar tareas como crear una conexión a una BD que se mantendrá mientras el servlet se mantenga cargado y puede ser utilizada por cada petición. **ServletConfig** contiene los parámetros de inicialización que entrega el servidor al servlet.
- **getServletConfig()**: Retorna la configuración dada para la inicialización del servlet.
- **service(ServletRequest req, ServletResponse res)**: Este método es el que se llama cuando se recibe una petición de un cliente y en su implementación normal para HTTP verifica el tipo de solicitud GET,

POST, etc. y la redirige a los métodos respectivos. En general no es necesario reimplementar este método.

- **destroy()**: Este método es llamado por el servidor para indicar que el servlet será destruido. Es llamado sólo una vez y uno debe encargarse de esperar por posibles peticiones en curso. Ver el ciclo de vida.

public abstract interface ServletConfig: Contiene los parámetros que entrega el servidor al servlet para ser inicializado que pueden ser dados por el administrador a través de un archivo de configuración. Entre sus métodos están:

- **getInitParameter(String name)**: Que retorna el valor del parámetro dado en *name*.
- **getServletContext()**: Que retorna un objeto **ServletContext** que guarda la información referente al servidor.

public abstract interface ServletContext: Contiene métodos que sirven para comunicar un servlet con el servidor que lo contiene.

- **getMimeType(String file)**: Retorna el tipo MIME definido en el servidor para el archivo dado.

public abstract class GenericServlet implements Servlet, ServletConfig: Define un servlet genérico independiente del protocolo. Además de implementar alguno de los métodos de las interfaz crea otros:

- **log(String msg)**: Escribe en la consola del servidor el mensaje junto con el nombre del servlet.

public abstract class HttpServlet extends GenericServlet: Es la clase de la cual se debe extender para crear un servlet HTTP. De la clase que extiende obtiene los métodos ya definidos además de los cuales define:

- **doGet(HttpServletRequest req, HttpServletResponse resp)**: Es el método llamado para procesar información que haya sido enviado con el método GET. Este método es llamado concurrentemente para cada cliente por lo que hay que estar atento por posibles variables compartidas que causen problemas.
- **doPost(HttpServletRequest req, HttpServletResponse resp)**: Ídem al anterior pero para el método POST, en general se implementa sólo un método y el otro lo referencia.

public abstract interface ServletRequest: Permite obtener información del cliente que no depende del protocolo, por ejemplo:

- **getRemoteAddr()**: Retorna la IP del cliente.
- **getParameter(String name)**: Retorna el valor del parámetro *name* dado por el cliente.
- **getInputStream()**: Sirve para crear un canal de comunicación para obtener dados binarios.

public abstract interface HttpServletRequest extends ServletRequest: Permite obtener del cliente la información que es dependiente del protocolo, en este caso HTTP. Entre sus métodos están:

- **getHeader(String name)**: Permite obtener el valor de los Headers de HTTP con que fue llamado el servlet.
- **getCookies()**: Retorna un arreglo que contiene todas las *cookies* que el cliente envía al servlet.
- **getSession()**: Retorna la sesión en la cual se encuentra el cliente.

public abstract interface HttpSession: Permite identificar al mismo usuario a través de distintos servlets. En general se implementa guardando una *cookie* en el cliente la cual es recuperada por el servidor para reasignar su sesión.

- **setAttribute(String name, Object value)**: Permite compartir un objeto cualquiera entre distintos servlets para el mismo usuario.
- **getAttribute(String name)**: Permite obtener un objeto previamente introducido a las sesión.
- **setMaxInactiveInterval(int interval)**: Permite definir un tiempo máximo para el cual la sesión será válida. Si transcurre ese tiempo y el usuario no ha dado respuesta el servidor borrará la sesión del usuario el cual se convertirá en un cliente recién ingresado.

public abstract interface ServletResponse: Define un objeto para permitir a un servlet enviar una respuesta al cliente.

- **setContentType(String type)**: Permite definir el tipo de respuesta que se le dará al cliente. Si se retornará una página web deberá ser *text/html.*
- **getWriter()**: Retorna un objeto Writer para poder enviar respuestas de texto.
- **getOutputStream()**: Retorna un objeto ServletOutputStream que permite enviar respuestas binarias al cliente.

public abstract interface HttpServletResponse extends ServletResponse: Permite enviar al cliente respuestas específicas del protocolo HTTP.

- **addCookie(Cookie cookie)**: Para definir nuevas *cookies* en el cliente.
- **setHeader(String name, String value)**: Para definir un header HTTP a enviar al cliente.

- **sendRedirect(String location)**: Envía un mensaje al cliente para redireccionar la respuesta a la dirección señalada.

public abstract interface SingleThreadModel: Esta interfaz no tiene métodos ya que sólo se utiliza para señalar al servidor que el servlet manejará sólo un requerimiento a la vez. Cualquier otra petición concurrente queda encolada. Es la solución simple y poco óptima para eliminar los problemas de concurrencia.

3. JSPs

3.1 ¿Qué es JSP?

Una JSP, acrónimo en inglés de JavaServer Pages, es una página Java en Servidor y es una plantilla para una página Web que emplea código Java para generar un documento HTML dinámicamente. Las JSP se ejecutan en un componente del servidor denominado ***contenedor de JSP***, y este las traduce o convierte a servlets de Java equivalentes y por lo tanto, lo que se puede hacer con un servlet, también se puede hacer con una JSP.

3.2 ¿JSP o Servlets?

Primeramente mencionaremos las ventajas de un Servlet y JSP sobre los CGI y luego las Ventajas de los JSP sobre los Servlets.

3.2.1 Ventajas de los Servlets y JSP

- Tienen un mejor desempeño y capacidad de adaptación que los CGI, debido a que se conservan en la memoria y manejan múltiples subprocesos.
- No se requiere una configuración especial por parte del cliente.
- Soportan sesiones HTTP, lo que hace posible la programación de aplicaciones.
- Pueden acceder a la tecnología disponible en Java para manejar hilos o treads, sockets o trabajo en red, conectividad con bases de datos y todo esto sin las limitaciones de los applets del cliente.

3.2.2 Ventajas de los JSP

- Se compilan automáticamente cuando sea necesario
- Su ubicación en el espacio común de documentos del servidor Web permiten ubicarlas más fácilmente que a los servlets
- Las páginas JSP son similares a las de HTML, por lo tanto son más compatibles con las herramientas de desarrollo de Web (DreamWeaver, FrontPage, etc.).

3.3 Algunos servidores Web con soporte para JSP's

Software	Desarrollado por la empresa	Sitio Web	Comentario
Apache Tomcat	The Apache Software Foundation	http://www.apache.org	Es uno de los desarrollos de software para manejo de servlets más popular. Actualmente esta en la versión 4.0 y es el resultado del proyecto Yakarta.
Java Web Server	Sun Microsystems	http://wwws.sun.com/software/jwebserver	Es el servidor Web de Sun desarrollado totalmente en Java. Actualmente en la versión 1.1.3. La empresa Sun anunció en 1999 que ya no esta en desarrollo activo porque se esta concentrando en el servidor Netscape/I-Planet, pero aún puede obtener una versión gratuita sin vencimiento para propósitos educativos.
JRun	Macromedia products	http://www.macromedia.com/es/software/jrun/	Es un servidor Web y además permite ejecutar servlets como si fuese un añadido del IIS, Apache, Nescape, etc. Actualmente esta en la vesión 3.0.
Resing server	Macromedia products	http://www.macromedia.com/es/software/jrun/	Es un servidor Web y además permite ejecutar servlets como si fuese un añadido del IIS, Apache, Nescape, etc. Actualmente esta en la vesión 3.0.
Resin Server 2.1	Caucho Technology	http://www.caucho.com/	Es un servidor comercial de la empresa Caucho developer. Puede utilizarse para desarrollo de JSP's, para desarrolladores o estudiantes. También puede adquirirse la licencia para uso comercial mediante pago.

3.4 Arquitectura JSP

3.4.1 Mecanismo de funcionamiento de una JSP

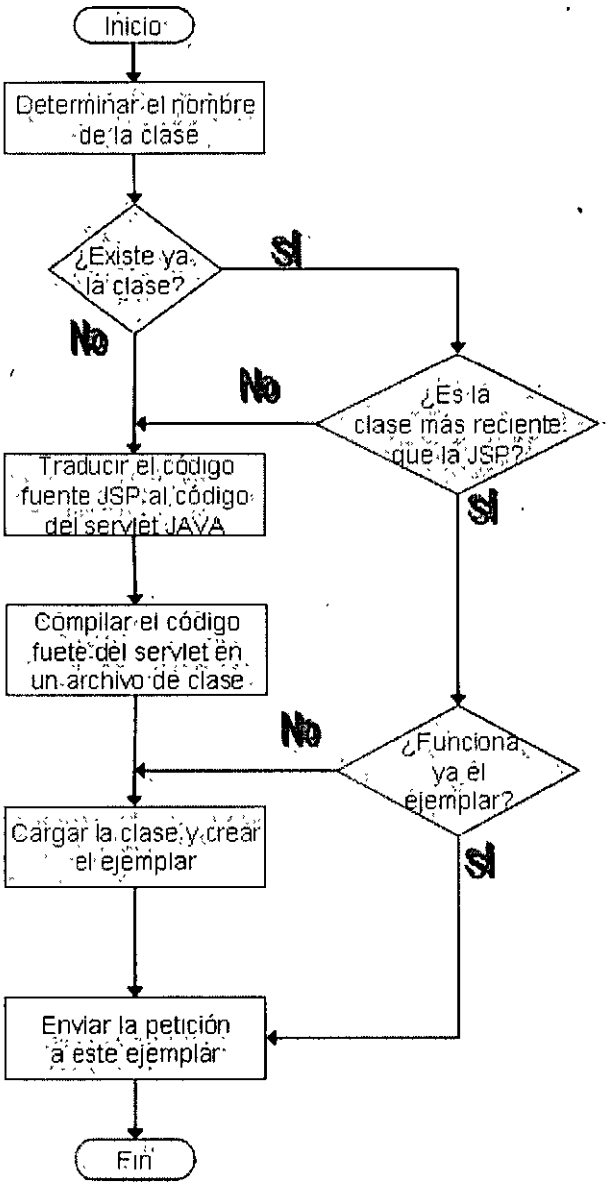
Una JSP atraviesa por etapas de evolución de tres pasos en su código:

1. **Código fuente JSP.** Es escrito por el programador o desarrollador de JSP. Está en un archivo de texto con extensión **.jsp** y se compone de una mezcla de código HTML, instrucciones en lenguaje Java, directivas JSP y acciones que describen cómo generar una pagina Web para responder a una solicitud por parte del cliente.
2. **Código fuente Java.** El contenedor de JSP traduce el código fuente JSP a código fuente de un servlet Java equivalente. Este código fuente se guarda en un área de trabajo y puede ser útil en el proceso de depuración de errores.
3. **Clase Java compilada.** Como sucede con cualquier otro programa de Java, el servlet generado se compila en byte code (código de bytes) resultando en un archivo **.class** que está listo para ser cargado y ejecutado por el servidor.

3.4.2 Gestión de las JSP por el contenedor

El contenedor de las JSP gestiona automáticamente cada una de éstas etapas de la página JSP. Por ejemplo, como respuesta a una petición HTTP, el contenedor comprueba si el archivo fuente **.jsp** ha tenido modificaciones desde que el código fuente **.java** se compiló por última vez.

También el contenedor determina primeramente el nombre de la clase correspondiente al archivo **.jsp**. Si la clase no existe o es anterior al archivo **.jsp** esto significa que el código fuente JSP ha cambiado desde que fue compilado por última vez. Luego entonces, el contenedor crea el código fuente Java para un servlet actualizado equivalente y lo compila. Si no hay aún una instancia o ejemplar del servlet en funcionamiento, el contenedor carga la clase servlet y crea un ejemplar. Finalmente, el contenedor lanza un subproceso para que gestione la petición HTTP actual del ejemplar encargado.



3.5 Incorporar contenido Java en una página HTML.

En una archivo .jsp se pueden tener elementos JSP, datos de plantilla fijos o cualquier combinación de ambos. Los elementos JSP son instrucciones dadas al contenedor de JSP sobre el código a generar y sobre cómo debe operar. Estos elementos tienen etiquetas específicas de comienzo y fin que los identifican en el proceso de compilación de JSP.

Los datos plantilla (generalmente HTML) son todo aquello que el contenedor de JSP no reconoce. Estos pasan a través del contenedor sin sufrir modificaciones, así que el HTML finalmente generado contiene los datos de plantilla tal y como estaban codificados en el archivo .jsp.

Hay tres tipos de elementos JSP:

- Elementos de secuencias de comandos (scripts) que incluyen expresiones, scriplets y declaraciones.
- Directivas.
- Acciones

3.5.1 Declaraciones.

Una declaración permite notificar al intérprete de Java que se va a definir nuevas variables o métodos en el archivo de la clase generada. Las declaraciones contienen instrucciones o sentencias en lenguaje Java con la sintaxis siguiente:

<%! sentencia; [sentencias; ...] %>

Las secciones de la declaración se pueden usar para declarar clases o variables de instancia, métodos o clases internas. Para declarar tanto una variable, como un método, se utiliza el símbolo "!". Las declaraciones son creadas e iniciadas cuando el usuario accede a la página JSP y su ámbito es de tipo "class", es decir, que están disponibles en toda la clase que genera después de la solicitar una página JSP. Cualquiera declaración hecha al comienzo de una página puede ser utilizada al final de la misma.

3.5.2 Expresiones.

Una expresión en una página JSP es un pequeño fragmento de código (scriptlet) que devuelve un resultado (salida) por pantalla. Su sintaxis es:

<%= expr %>

donde **expr** es cualquier expresión de Java válida. La expresión puede tomar cualquier valor como un dato, mientras éste se pueda convertir en una cadena. Esta conversión se efectúa generalmente con una instrucción out.print() o puede ser evaluada como un java.util.String, o bien directamente mediante la invocación al método toString() o del método String.valueOf().

Una expresión puede ser algo tan sencillo como un valor numérico primitivo como:

** Una operación sencilla: 3 + 4 = <%= 3 + 4 %> **

o puede ser una expresión más compleja con llamadas a métodos:

<%= new java.text.SimpleDateFormat("d MMM, yyyy").format(new java.util.Date()) %>

3.5.3 Scriptlets

Un scriptlet es un conjunto de instrucciones o sentencias de Java incluidas en una página HTML. Estas instrucciones se distinguen del HTML porque están colocadas entre los marcadores <% y %> para que el intérprete de JSP sepa que debe procesar todo el código que se encuentre dentro de esas etiquetas. Su sintaxis es:

<% sentencias; [sentencias; ...] %>

3.6 Directivas

Son etiquetas que se utilizan en una página JSP cuya principal característica es incluir el símbolo @ en su sintaxis, que tiene la siguiente forma:

<%@ nombre_directiva [atributo_i = "valor_i"] %>

Su ámbito es de tipo page (página), esto es, su acción sólo ocurrirá en la página en la que han sido utilizadas. además no devuelven ningún valor al cliente como las expresiones o los scriptlets, salvo lo generado por su ejecución. La especificación JSP 1.1 describe tres directivas estándar disponibles en todos los entornos JSP que son: page, include y taglib

3.6.1 Directiva page.

Se utiliza para definir atributos globales que deben ser aplicados a la página JSP completa, y a cualquier archivo , excepto los de contenido dinámico, que se haya incluido en ésa página con la directiva include o la acción jsp:include. Su sintaxis es la siguiente:

<%@ page [languaje= "java"] [extends= "paquete.clase"]
[import="{paquete.clase | paquete.*},..."]
[session="true | false"] {buffer="none | 8kb | tamaño kb"]
[autoFlush="true | false"] [isThreadSafe="true | false"]
[info="mensaje"] [errorPage="URL relativa"]
[contentType="{tipoMime [; charset=juegoCaracteres]
| text/html ; charset=ISO-8859-1}"] [isErrorPage="true | false"]
[pageEncoding="{juegoCaracteres | ISO-8859-1}"]

Los términos opcionales se sitúan entre corchetes y la barra vertical indica que cualquiera de los valores puede ser utilizado. Los atributos de la directiva se detallan a continuación:

languaje ="java"

Define el lenguaje que se utilizará en los scriptlets o trozos de código que se colocan en la página JSP o en cualquier archivo que se incluye en la página mediante la directiva o la acción correspondiente. Actualmente, la especificación JSP 1.2 permite asignar solamente el valor java para este atributo.

extends="paquete.clase"

Especifica el nombre de una clase, totalmente calificada, que se pueda extender en el archivo JSP. Notifica a la clase generada después de solicitar la página JSP si hereda de una superclase. Por defecto, este valor no es utilizado.

import="{paquete.clase | paquete.*},..."

Permite especificar una lista de paquetes o clases, separados por comas, que pueden ser invocados en los scriptlets, expresiones, declaraciones, etiquetas que se utilicen dentro de la página JSP. Este atributo debe aparecer antes de cualquiera de las clases que se importen.

Los paquetes que se importan por default son:
java.lang.*, javax.servlet.*, javax.servlet.jsp.* y javax.servlet.http.*;

sesion="true | false"

Cuando se realiza una petición por parte de un cliente al servidor Web, éste le asigna un identificador a esa sesión para poder referirse a la conexión HTTP que se ha establecido. Si se indica true (el valor por defecto si no se especifica el atributo en la directiva) el objeto session se puede utilizar y se refiere a la sesión actual o a cualquier nueva sesión que se establezca, si se indica false como valor de este atributo, el objeto session no se podrá utilizar en la página JSP.

buffer="none | 8kb | tamañokb"

Especifica el tamaño del buffer utilizado por un objeto implícito **out** para enviar el resultado de la salida compilada de la página JSP al navegador cliente. Puede indicarse que no se use, en cuyo caso la salida es enviada directamente a través del objeto **PrintWriter**; o un tamaño específico, siendo el valor por defecto 8 kb.

autoFlush="true | false"

Controla el buffer de salida. Si el valor es true, su valor por defecto, cuando el buffer de salida esté completo, se descargará hacia el navegador; si su

valor es false el buffer no se descarga, con lo que se genera una excepción de buffer completo cuando el buffer esté lleno. Este atributo no puede ser false cuando el valor del atributo buffer es none.

isThreadSafe="true | false"

Indica al motor JSP cómo se debe implementar la seguridad de las tareas. Si el valor es true, el que toma por omisión, el motor puede enviar múltiples solicitudes concurrentes a la página JSP; es decir, varias tareas diferentes podrían acceder a la página JSP, por lo que es responsabilidad del desarrollador sincronizar los métodos para proporcionar seguridad entre las tareas.

info="mensaje"

Indica un mensaje de texto, generalmente de descripción o información de copyright, que será incorporado a la página JSP y que, posteriormente, puede ser recuperado invocando al método getServletInfo().

errorPage="URL relativa"

Especifica una página JSP a la que se desvía la ejecución cuando se produce una excepción en la página que no esté capturada. Si la ruta comienza con una barra inclinada (/), la URL será relativa a la raíz de documentos del servidor web, por lo que éste es el encargado de la resolución completa de la dirección de la página de error; mientras si no comienza con la barra inclinada, la ruta estará referenciada al directorio actual que de fine la directiva page.

contentType="{tipoMime [; charset=juegoCaracteres] | text/html ; charset=ISO-8859-1}"

Indica el tipo mime y la modificación del conjunto de caracteres que se enviará en la respuesta que se genera hacia el cliente. Se puede utilizar cualquier tipo mime y juego de caracteres soportado por el motor JSP. Por defecto, se envía la página HTML con el conjunto de caracteres europeo normal.

isErrorPage="true | false"

Este atributo indica si la página JSP actual es una página de error. Si el valor es true, la página tendría acceso al objeto implícito exception, que contiene una referencia a la excepción que haya lanzado el motor JSP.

pageEncoding="{juegoCaracteres | ISO-8859-1}"

Indica el conjunto de caracteres que se utiliza en la propia página JSP. Por omisión, en caso de que se haya especificado un valor diferente para el atributo contentType, ése será el valor de pageEncoding, o bien ISO-8859-1 en cualquier otro caso.

3.6.2 Directiva include.

Permite la inclusión en una página JSP en tiempo de compilación, de una página HTML, archivo de código Java, archivo de texto u otra página JSP. La página final que va a procesar el motor JSP esta conformada por la página base más el contenido del recurso que se haya incluido. Esta directiva puede ir situada en cualquier lugar de la página JSP. Al usar esta directiva se recomienda especial cuidado en la colocación de las etiquetas HTML <html></html>, <body></body>, etc., porque el archivo incluido podría entrar en conflicto con las etiquetas similares del archivo JSP original, por lo que hay que verificar que estén correctamente anidadas y no duplicadas.

3.7 Objetos implícitos

Existen unos objetos implícitos o integrados que están disponibles en todo momento para el programador de páginas JSP. No hay necesidad de instanciarlos, ya que están disponibles en el contenedor JSP. Estos objetos existen para hacer más sencilla la programación de páginas JSP.

3.7.1 Objeto page

Es una instancia de la clase *java.lang.Object*, y su ámbito es de tipo *page*. Representa la instancia de una clase generada a partir de una página JSP, es decir la página JSP actual. Cuando se quiere hacer referencia a esta página JSP se utiliza la palabra *this*.

3.7.2 Objeto config

Es una instancia de la clase *javax.servlet.ServletConfig* y maneja todo lo relacionado a la configuración del servlet generado al solicitar la página JSP. Su ámbito de funcionamiento es de tipo *page*.

3.7.3 Objeto request

Es una instancia de *javax.servlet.ServletRequest* y contiene información de datos enviados al servidor a través de una página Web. Por ejemplo el método *getParameter()* del objeto *request* recoge valores enviados por medio de un formulario o una URL. Así, puede recuperarse los valores o contenido de los campos de un formulario de una página Web, ejemplo:

```
<%
    String nombre =  request.getParameter("nombre");
%>
```

3.7.4 Objeto sesión

Es una instancia de la clase *javax.servlet.http.HttpSesion*. El cometido de este objeto es manejar todas las acciones relacionadas a la sesión del usuario, por lo tanto su ámbito de utilización es de tipo sesión. Una sesión es creada automáticamente (a menos que se especifique lo contrario) cuando un usuario solicita una página JSP, y de esta manera podemos almacenar información relativa a ese usuario.

Si por ejemplo deseamos almacenar un objeto en la sesión, para posteriormente poder recuperarlo hacemos:

```
<%
    HttpSession  unaSesion = request.getSession();
    unaSesion.setAttribute(usuario,"pancho_lopez");
%>
```

3.7.5 Objeto application

Es una instancia de la clase java.servlet.ServletContext, y su ámbito de utilización es de tipo application. Representa la aplicación Web en el que se está ejecutando la página JSP.

4. Beans de Entidad

4.1 Definición

Un bean de entidad está pensado para representar la lógica de negocio de una entidad existente en un almacenamiento persistente, como una base de datos relacional. Los beans de entidad comparten algunas cualidades que podría encontrar en una base de datos relacional, por ejemplo:

- Los beans de entidad permiten compartir accesos: los beans podrían compartirse entre varios clientes y la concurrencia la maneja el contenedor.
- Los beans de entidad tienen claves primarias: Existen clases Primary-key para identificar un ejemplar de un bean de entidad. La clave primaria contiene toda la información necesaria para encontrar un bean almacenado.
- Los beans de entidad podrían participar en relaciones: Se han presentado interfaces locales para manejar las relaciones entre beans.

4.2 Sentencias en JSPs

En las JSPs, existen las siguientes sentencias para usar estas propiedades de los beans de entidad en relación con la programación:

<jsp:usebean>	Obtiene la referencia de objeto (JavaBean) que existe en un ámbito específico y con un nombre específico. Dependiendo de la sintaxis puede instanciar un nuevo objeto si no lo encuentra.
<jsp:getproperty>	Obtiene el valor de una propiedad (atributo) de un JavaBean y la incluye directamente en la salida generado para su envío al cliente.
<jsp:setproperty>	Asigna un valor a una propiedad (atributo) de un JavaBean cogiéndolo de un parámetro de la petición (request) del cliente o asignándoselo directamente.

JAVASERVER PAGES™ (JSP) SYNTAX

JSP 1.1

Element	Description	JSP Syntax
	Legend	All tags are case sensitive. A pair of single quotes is equivalent to a pair of double quotes. Spaces are not allowed between an equals sign and an attribute value. plain text = required **bold** = default *italics* = user-defined | = or [ ] = optional { } = required choice ... = list of items + = can repeat
HTML Comment	Creates a comment that is sent to the client in the viewable page source.	<!-- *comment* [<%= *expression* %>] -->
Hidden Comment	Documents the JSP file, but is not sent to the client.	<%-- *comment* --%>
Declaration	Declares variables or methods valid in the page scripting language.	<%! *declaration; [declaration;]+ ...* %>
Expression	Contains an expression valid in the page scripting language.	<%= *expression* %>
Scriptlet	Contains a code fragment valid in the page scripting language.	<% *code fragment of one or more lines* %>
Include Directive	Includes a static file, parsing the file's JSP elements.	<%@ include file="*relativeURL*" %>
Page Directive	Defines attributes that apply to an entire JSP page.	<%@ page [language="**java**"] [extends="*package.class*"] [import="{*package.class* | *package.**}, ... "] [session="**true** | false"] [buffer="none | **8kb** | *size*kb"] [autoFlush="**true** | false"] [isThreadSafe="**true** | false"] [info="*text*"] [errorPage="*relativeURL*"] [contentType="{*mimeType* [; charset=*characterSet*] | **text/html ; charset=ISO-8859-1**}"] [isErrorPage="true | **false**"] %>
Taglib Directive	Defines a tag library and prefix for the custom tags used in the JSP page.	<%@ taglib uri="*URIToTagLibrary*" prefix="*tagPrefix*" %>
<tagPrefix:name>	Accesses a custom tag's functionality.	*<tagPrefix:name attribute="value"+ ... />* *<tagPrefix:name attribute="value"+ ... > other tags and data </tagPrefix:name>*

Sun microsystems

For more information visit: http://java.sun.com/products/jsp

JAVA

JAVASERVER PAGES™ (JSP) SYNTAX

JSP 1.1

Element	Description	JSP Syntax
<jsp:forward>	Forwards a client request to an HTML file, JSP file, or servlet for processing.	<jsp:forward page="{*relativeURL* | <%= *expression* %> }" { /> | [<jsp:param name="*parameterName*" value="{*parameterValue* | <%= *expression* %>}" />]+ </jsp:forward> }
<jsp:getProperty>	Gets the value of a Bean property so that you can display it in a result page.	<jsp:getProperty name="*beanInstanceName*" property="*propertyName*" />
<jsp:include>	Includes a static file or sends a request to a dynamic file.	<jsp:include page="{*relativeURL* | <%= *expression* %>}" flush="true" { /> | [<jsp:param name="*parameterName*" value="{*parameterValue* | <%= *expression* %>}" />]+ </jsp:include> }
<jsp:plugin>	Downloads plug-in software to the Web browser to execute an applet or Bean.	<jsp:plugin type="bean|applet" code="*classFileName*" codebase="*classFileDirectoryName*" [name="*instanceName*"] [archive="*URIToArchive, ...*"] [align="**bottom**|top|middle|left|right"] [height="*displayPixels*"] [width="*displayPixels*"] [hspace="*leftRightPixels*"] [vspace="*topBottomPixels*"] [jreversion="*JREVersionNumber* | 1.1"] [nspluginurl="*URLToPlugin*"] [iepluginurl="*URLToPlugin*"] > [<jsp:params> [<jsp:param name="*parameterName*" value="{*parameterValue* | <%= *expression* %>}" />]+ </jsp:params>] [<jsp:fallback> *text message for user* </jsp:fallback>] </jsp:plugin>
<jsp:setProperty>	Sets a property value or values in a Bean.	<jsp:setProperty name="*beanInstanceName*" { property="*" | property="*propertyName*" [param="*parameterName*"] | property="*propertyName*" value="{*string* | <%= *expression* %>}" } />
<jsp:useBean>	Locates or instantiates a Bean with a specific name and scope.	<jsp:useBean id="*beanInstanceName*" scope="page|request|session|application" { class="*package.class*" [type="*package.class*"] | type="*package.class*" | beanName="{*package.class* | <%= *expression* %>}" type="*package.class*" } { /> | > *other elements* </jsp:useBean> }

Implicit Objects	Type	Scope	Some Useful Methods (see class or interface for others)
request	Subclass of *javax.servlet.ServletRequest*	Request	getAttribute, getParameter, getParameterNames, getParameterValues
response	Subclass of *javax.servlet.ServletResponse*	Page	*Not typically used by JSP page authors*
pageContext	*javax.servlet.jsp.PageContext*	Page	findAttribute, getAttribute, getAttributesScope, getAttributeNamesInScope
session	*javax.servlet.http.HttpSession*	Session	getId, getValue, getValueNames, putValue
application	*javax.servlet.ServletContext*	Application	getMimeType, getRealPath
out	*javax.servlet.jsp.JspWriter*	Page	clear, clearBuffer, flush, getBufferSize, getRemaining
config	*javax.servlet.ServletConfig*	Page	getInitParameter, getInitParameterNames
page	*java.lang.Object*	Page	*Not typically used by JSP page authors*
exception	*java.lang.Throwable*	Page	getMessage, getLocalizedMessage, printStackTrace, toString

Is this heaven? No, it's the JavaServer Pages technology. Printed in USA 2/2000