E.T.S. INGENIERÍA INFORMÁTICA

Gestión de la Información

UNIVERSIDAD DE MÁLAGA

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

Tema 6

Aplicaciones Java + JDBC

José Luis Pastrana Brincones (pastrana@lcc.uma.es)

JDBC

¿Qué es JDBC?

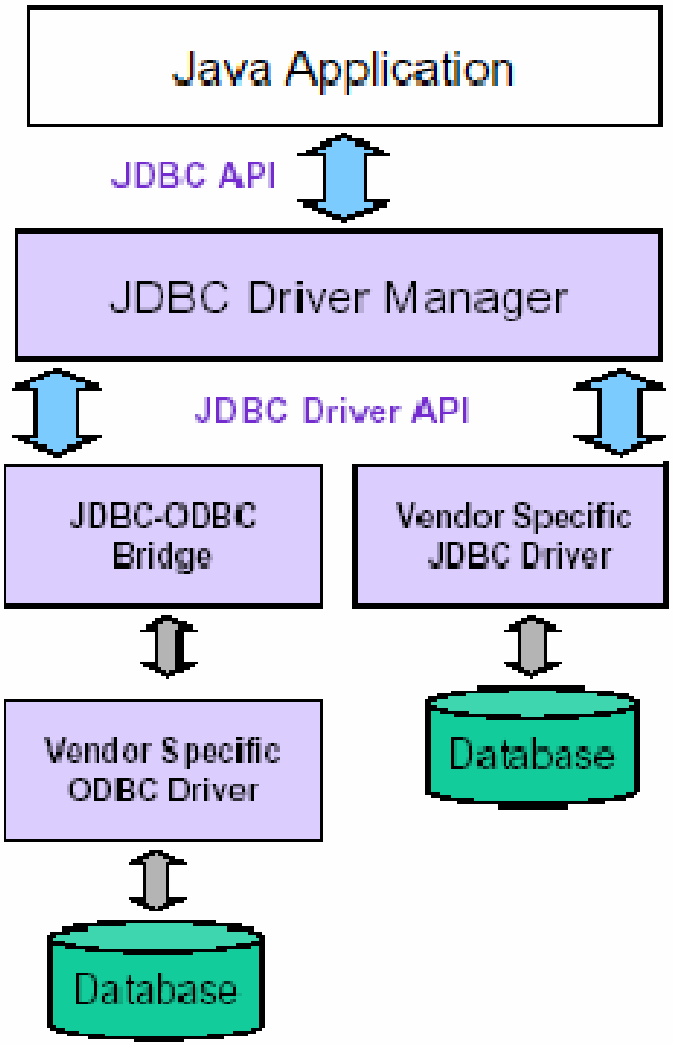
- JDBC (Java DataBase Connectivity) consiste en una API de alto nivel y diferentes *drivers cada uno para conectarse a una base de datos* distinta.

- JDBC proporciona una interfaz estándar de acceso a bases de datos relacionales.

- JDBC es independiente de dónde se encuentre el cliente y dónde esté el servidor.

JDBC

- Arquitectura General



JDBC

- Un programa Java se conecta a una base de datos con JDBC realizando las siguientes operaciones:

1. Importación de paquetes
2. Carga del driver JDBC
3. Conexión con la base de datos
4. (Opcional) Averiguar las capacidades de la base de datos
5. (Opcional) Obtención de metadatos propios del esquema al que nos hemos conectado
6. Construcción de la sentencia SQL y ejecución
7. Procesamiento de resultados, si los hay
8. Cierre de la sentencia y del cursor, si lo hay
9. Cierre de la conexión

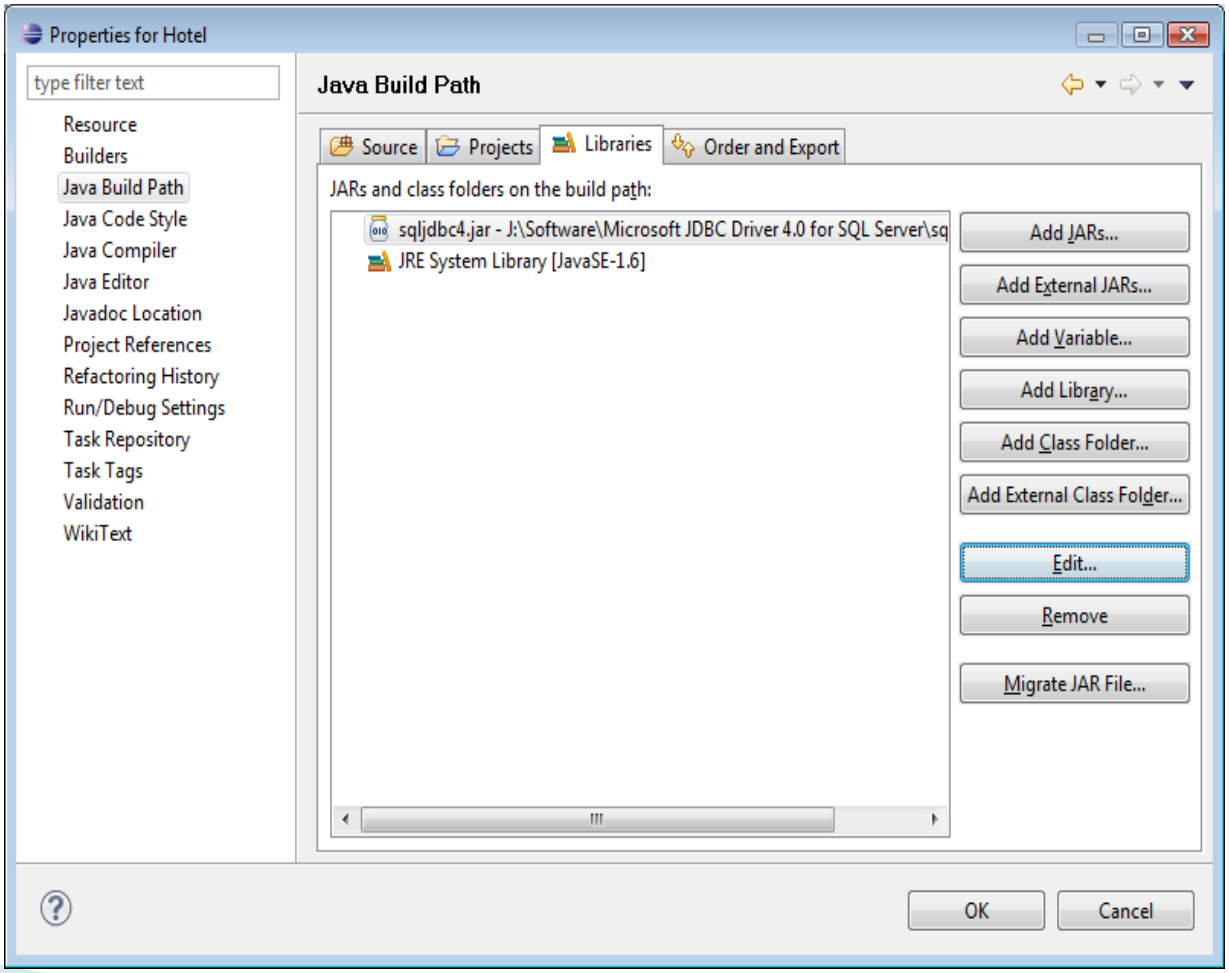
JDBC

- El controlador JDBC de Microsoft SQL Server es un controlador compatible con Java Database Connectivity (JDBC) 4.0 que proporciona un acceso confiable a los datos de las bases de datos de Microsoft SQL Server.
- Puede descargarse desde la página de la asignatura en el campus virtual o desde http://msdn.microsoft.com/es-es/sqlserver/aa937724.aspx

JDBC

Para usar el Driver de JDBC-SQL Server hay que:

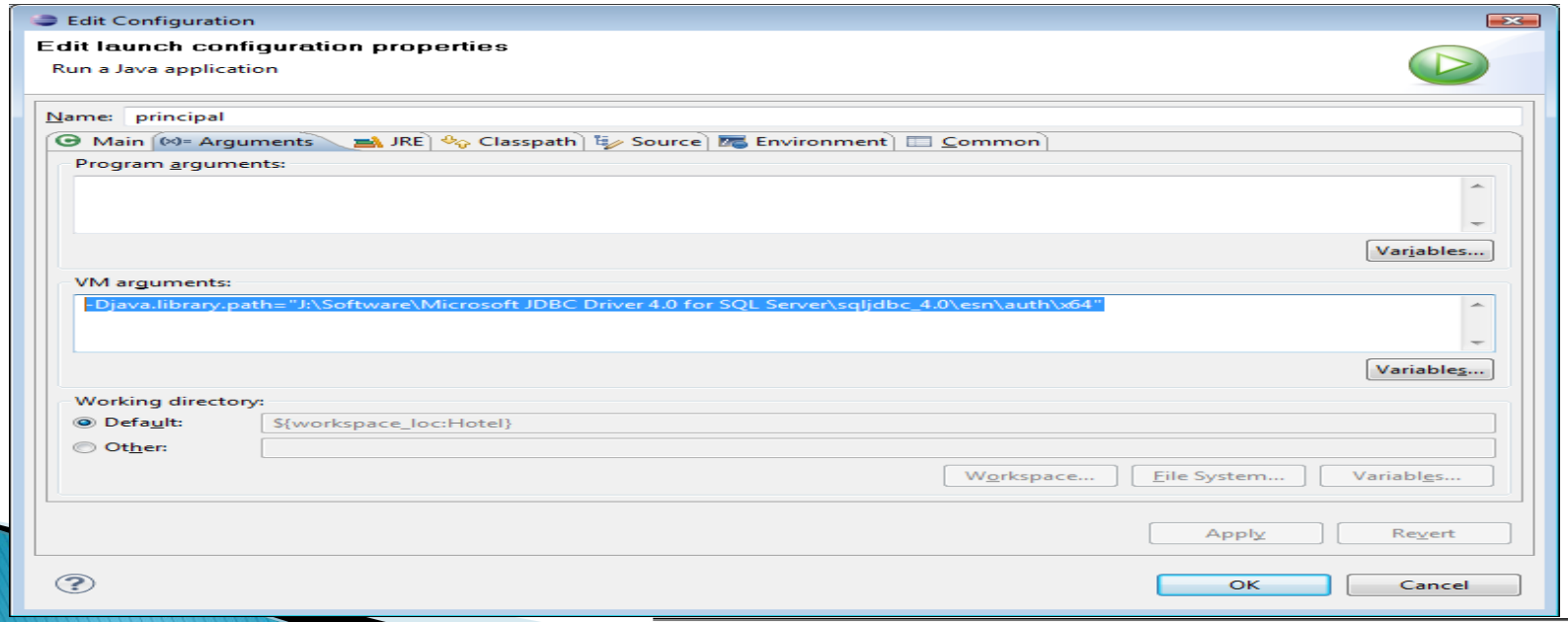
- Agregar `sqljdbc.jar` ó `sqljdbc4.jar` en el path de compilación de nuestro proyecto java



JDBC

- Si usamos autentificación de windows: Incluir la librería correspondiente (x86/x64) en el entorno de ejecución de nuestra máquina virtual de java:

```
-Djava.library.path="…\Microsoft JDBC Driver
  4.0 for SQL Server\sqljdbc_4.0\esn\auth\x64"
```



JDBC

- JDBC 1.0 permite accesos más o menos básicos a una base de datos.
- JDBC 2.0 permite muchas más cosas:
  - Los cursores (llamados `resultset`), se pueden recorrer de arriba abajo o viceversa, incluso cambiando el sentido.
  - Se admite un mayor número de tipos de datos.
  - Pool de conexiones.
  - Transacciones distribuidas. Paquetes `javax.sql.*` en el lado cliente.
- El JDK 1.2 es compatible con JDBC 2.0 a través del paquete `java.sql.*`.

JDBC

- JDBC 3.0 permite algunas cosas más:
  - Al llamar a un procedimiento almacenado, los parámetros pueden pasarse por nombre en vez de por posición.
  - Si una tabla tiene un campo autonumérico, permite averiguar qué valor se le ha dado tras una inserción.
  - Se proporciona un pool de conexiones automático.
  - Se permite el uso de SAVEPOINT.
- JDBC 4.0 aún añade más cosas:
  - No es necesario cargar el driver para realizar la conexión. El `DriverManager` lo hace de manera inteligente y automática.
  - Soporta el tipo ROWID mediante la clase `RowId`.
  - Mejora en la gestión de excepciones: `SQLException` tiene nuevas subclases y las excepciones múltiples pueden recorrerse mediante un bucle.
  - Soporte para caracteres nacionales.

JDBC

- Usar JDBC con SQL Server

1. Importación de paquetes

```
import java.sql.*;
import com.microsoft.sqlserver.jdbc.*;
```

2. Carga del driver JDBC El driver puede cargarse de dos formas:
   - Mediante la biblioteca de clases `sqljdbc.jar`, primero las aplicaciones deben registrar el controlador del modo siguiente:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

   - Una vez cargado el controlador, puede establecer una conexión con una URL de conexión y el método **getConnection** de la clase **DriverManager**:

JDBC

- Ejemplo:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
String connectionUrl = "jdbc:sqlserver://localhost:1433;" +
      “databaseName=myDB;user=MyUserName;password=*****;";”
Connection con = DriverManager.getConnection(connectionUrl);
```

- Conexión a la base de datos con seguridad de windows

```
jdbc:sqlserver://localhost;databaseName=myDB;
integratedSecurity=true;
```

- Conexión en el puerto predeterminado con el servidor remoto:

```
jdbc:sqlserver://localhost:1433;databaseName=myDB;
integratedSecurity=true;
```

JDBC

- En la API 4.0 de JDBC, el método `DriverManager.getConnection` se ha mejorado para que se carguen los controladores JDBC automáticamente. Por tanto, no es necesario que las aplicaciones llamen al método `Class.forName` para registrar o cargar el controlador cuando se use la biblioteca de clases `sqljdbc4.jar`.

- Cuando se llama al método `getConnection` de la clase `DriverManager`, se encuentra un controlador apropiado en el conjunto de controladores JDBC. El archivo `sqljdbc4.jar` incluye el archivo "`META-INF/services/java.sql.Driver`", que contiene el `com.microsoft.sqlserver.jdbc.SQLServerDriver` como un controlador registrado. Las aplicaciones existentes, que actualmente cargan los controladores usando el método `Class.forName`, seguirán trabajando sin modificación

JDBC

- Ejemplo:

```
SQLServerDataSource ds = new SQLServerDataSource();
ds.setIntegratedSecurity(true);
ds.setServerName(“localhost");
ds.setPortNumber(1433);
ds.setDatabaseName("hotel");
Connection con = ds.getConnection();
```

JDBC

Construcción de la sentencia SQL y ejecución

- Toda sentencia está asociada a una conexión.

```
Statement stmt = con.createStatement();
```

- Si la sentencia a ejecutar es un SELECT se ejecuta:

```
stmt.executeQuery("SELECT * FROM usuario;”);
```

- Si la sentencia a ejecutar es un INSERT, UPDATE o DELETE se ejecuta:

```
stmt.executeUpdate("DELETE FROM usuario WHERE
    Nombre=‘pepe';");
```

Procesamiento de resultados, si los hay

- Un **executeQuery devuelve un cursor,** implementado con un objeto de tipo **ResultSet.**

```
ResultSet rset = stmt.executeQuery("SELECT * FROM
   usuario;”);
```

JDBC

Cierre de la sentencia y del cursor, si lo hay

```
rset.close();
stmt.close();
```

Cierre de la conexión

```
con.close();
```

JDBC

- **Procesamiento del ResultSet**
- El procesamiento se divide en dos partes:
  1. Acceso a los registros obtenidos, de uno en uno.
  2. Obtención de los valores de las columnas, de una en una.
- El acceso a los registros se hace en base al método **next() de los ResultSet., que devuelve un valor** *booleano que indica si hay o no siguiente registro.*
- Las columnas pueden ser accedidas por nombre o por posición.
- Para ello se usa un método que depende del tipo de
- la columna obtenida: **getInt, getString, getFloat, etc.**

JDBC

- **Ejemplo Procesamiento del ResultSet.**

```
while (rset.next())
{
      System.out.println(
            rset.getString(“Nombre"); + "-" +
            rset.getString(“pwd") + "-" +
            rset.getInt(“rol“) );
}
```

JDBC

- **Procesamiento del ResultSet**
- Una vez procesador un `ResultSet` hay que cerrarlo, así como la sentencia que lo originó.
- Caso de no hacerlo se estarán ocupando cursores y bloques de memoria innecesarios, pudiendo producirse un error al superar el número máximo de cursores abiertos.
- Una vez que se haya acabado todo el procesamiento con la base de datos, se debe cerrar la conexión.

JDBC

- La conexión se abre una sola vez al comienzo de la sesión y se cierra una vez acaba la sesión. Una misma sesión puede ejecutar múltiples sentencias.
- Tanto los **ResultSet, las Statement y las Connection se cierran con close().**
- Los métodos relacionados con SQL pueden lanzar la excepción SQLException, por lo que todo ha de enmarcarse en un try-catch.

JDBC. Ejemplo Select

```
SQLServerDataSource ds = new SQLServerDataSource();
ds.setIntegratedSecurity(true);
ds.setServerName(“localhost");
ds.setPortNumber(1433);
ds.setDatabaseName("hotel");
Connection con = ds.getConnection();
Statement stmt = con.createStatement();
ResultSet rset = stmt.executeQuery("SELECT * FROM usuario WHERE Nombre='“
        + name + "';");
rset.next(); // Solo debe haber 1 resultado
System.out.println( name + "-" +rset.getString(“pwd") + "-"
        + rset.getInt(“rol“) );
rset.close(); stmt.close(); con.close();
```

JDBC. Ejemplo Insert.

```
SQLServerDataSource ds = new SQLServerDataSource();
ds.setIntegratedSecurity(true);
ds.setServerName(“localhost");
ds.setPortNumber(1433);
ds.setDatabaseName("hotel");
Connection con = ds.getConnection();
Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO usuario VALUES('"
     + n + "','" + p + "'," + r + ");");

stmt.close();
con.close()
```

JDBC. Ejemplo Update.

```
SQLServerDataSource ds = new SQLServerDataSource();
ds.setIntegratedSecurity(true);
ds.setServerName(“localhost");
ds.setPortNumber(1433);
ds.setDatabaseName("hotel");
Connection con = ds.getConnection();
Statement stmt = con.createStatement();
stmt.executeUpdate("UPDATE usuario SET pwd='“
   + pwd + "' WHERE Nombre='"+ Nombre + "';");
stmt.close();
con.close();
```

JDBC. Ejemplo Delete.

```
SQLServerDataSource ds = new SQLServerDataSource();
ds.setIntegratedSecurity(true);
ds.setServerName(“localhost");
ds.setPortNumber(1433);
ds.setDatabaseName("hotel");
Connection con = ds.getConnection();
Statement stmt = con.createStatement();
stmt.executeUpdate("DELETE FROM usuario WHERE Nombre='“
     + n + "';");
stmt.close();
con.close();
```

JDBC. Ejemplo Procedimiento Almacenado.

- Para trabajar con los datos de una base de datos de SQL Server con un procedimiento almacenado, el controlador JDBC de Microsoft SQL Server proporciona las clases:
  - `SQLServerStatement`
  - `SQLServerPreparedStatement`
  - `SQLServerCallableStatement.`
- La clase usada depende de si el procedimiento almacenado requiere los parámetros IN (entrada) u OUT (salida).

JDBC

- Si el procedimiento almacenado no requiere ningún parámetro IN u OUT, puede usar la clase SQLServerStatement;
- Si requiere solo parámetros IN, puede usar la clase SQLServerPreparedStatement.
- Si el procedimiento almacenado requiere parámetros IN y OUT, debe usar la clase SQLServerCallableStatement.
- Solo si el procedimiento almacenado requiere parámetros OUT, necesita usar la clase SQLServerCallableStatement.

JDBC

- Los procedimientos almacenados también pueden devolver recuentos de actualizaciones y múltiples conjuntos de resultados.
- Si usa el controlador JDBC para llamar a un procedimiento almacenado con parámetros, debe usar la secuencia de escape de SQL call junto con el método prepareCall de la clase `SQLServerConnection`.
- La sintaxis completa de la secuencia de escape call es la siguiente:
- `{[?=]call procedure-name[([parameter][,[parameter]]...)]}`

JDBC

- Si el procedimiento almacenado no requiere ningún parámetro IN u OUT.Ejemplo:

```
CREATE PROCEDURE GetContactFormalNames
AS
BEGIN
   SELECT TOP 10 Title + ' ' + FirstName + ' '
   + LastName AS FormalName
   FROM Person.Contact;
END
```

JDBC

```
public void executeSprocNoParams(Connection con)
{   try
    { Statement stmt = con.createStatement();
       ResultSet rs = stmt.executeQuery("{call  GetContactFormalNames}");
       while (rs.next())
      {
         System.out.println(rs.getString("FormalName"));
       }
       rs.close();
       stmt.close();
    }
    catch (Exception e) { e.printStackTrace(); }
}
```

JDBC

- Si requiere solo parámetros IN. Ejemplo:

```
public void executeSprocInParams(Connection con)
{
   try
   {
      PreparedStatement pstmt = con.prepareStatement(
            "{call dbo.uspGetEmployeeManagers(?)}");
      pstmt.setInt(1, 50);
      ResultSet rs = pstmt.executeQuery();
```

JDBC

```
while (rs.next())
{
  System.out.println("EMPLOYEE:");
  System.out.println(rs.getString("LastName") + ", " +
    rs.getString("FirstName"));
  System.out.println("MANAGER:");
  System.out.println(rs.getString("ManagerLastName") + ", " +
    rs.getString("ManagerFirstName"));
  System.out.println();
 }
 rs.close();
 pstmt.close();
}
catch (Exception e) { e.printStackTrace();}
}
```

JDBC

- Si sólo requiere parámetros IN y OUT. Ejemplo:

```
CREATE PROCEDURE GetImmediateManager
    @employeeID INT,
    @managerID INT OUTPUT
AS
BEGIN
    SELECT @managerID = ManagerID
    FROM HumanResources.Employee
    WHERE EmployeeID = @employeeID
END
```

JDBC

```
public void executeStoredProcedure(Connection con)
{
   try {
      CallableStatement cstmt = con.prepareCall(
            "{call dbo.GetImmediateManager(?, ?)}");
      cstmt.setInt(1, 5);
      cstmt.registerOutParameter(2,
            java.sql.Types.INTEGER);
      cstmt.execute();
      System.out.println("MANAGER ID: " +
    cstmt.getInt(2));
   }
   catch (Exception e) { e.printStackTrace(); }
}
```

JDBC

- Caso de llamar a una función, se hace:

```
prepareCall("{ ? = call nombre_proc(?, ?, ?)}")
```

- donde el 1er ? se trata como un parámetro de salida.
- Ejemplo:

```
CallableStatement  cstmt =
  con.prepareCall ("{ ? = call getRol(?,?)}");
cstmt.registerOutParameter(1,java.sql.Types.INTEGER);
cstmt.setString(2, n);
cstmt.setString(3, p);
cstmt.execute();
r = cstmt.getInt(1);
```

JDBC. Procesamiento básico de transacciones

- Por defecto, JDBC hace un `COMMIT` automático tras cada `INSERT`, `UPDATE`, o `DELETE` (excepto si se trata de un procedimiento almacenado).
- Este comportamiento se puede cambiar mediante el método `setAutoCommit(boolean)` de la clase `Connection`.
- Para hacer un `COMMIT`, la clase `Connection` posee el método `commit()`.
- Para hacer un `ROLLBACK`, la clase `Connection` posee el método `rollback()`.

JDBC. Procesamiento básico de transacciones

- Si se cierra la conexión sin hacer `commit()` o `rollback()` explícitamente, se hace un `COMMIT` automático, aunque el `AUTO COMMIT` esté a `false`.

- También se permite el establecimiento y recuperación de puntos de salvaguarda con `Savepoint setSavepoint(String nombre)`

y

`rollback(Savepoint sv)`

JDBC. Controlar metadatos

- El controlador JDBC de Microsoft SQL Server se puede usar para trabajar con los metadatos de una base de datos de SQL Server de distintos modos. El controlador JDBC se puede usar para obtener metadatos sobre la base de datos, un conjunto de resultados o los parámetros.
- El controlador JDBC proporciona tres clases para recuperar metadatos de una base de datos de SQL Server:
  - `SQLServerResultSetMetaData`, que se usa para devolver información sobre el conjunto de resultados.
  - `SQLServerDatabaseMetaData`, que se usa para devolver información sobre la base de datos conectada.
  - `SQLServerParameterMetaData`, que se usa para devolver información sobre los parámetros de instrucciones preparadas e invocables.

JDBC. Metadatos de ResultSet

- Cuando un `ResultSet` recoge datos de una consulta desconocida (p.ej., introducida por teclado), no se sabe qué campos obtener.
- Para saber la estructura de una consulta se utiliza la clase `SQLServerResultSetMetaData`.
- Sus funciones más interesantes son:
  - ```
    int getColumnCount();
    // Número de campos por registro.
    ```
  - ```
    String getColumnName(int i)
    // Nombre de la columna i-ésima.
    ```
  - ```
    int getColumnType(int i)
    // Tipo de la columna i-ésima.
    ```
- Los tipos de las columnas vienen definidos por las constantes de `java.sql.Types`.
    ```

JDBC. Metadatos de ResultSet

- Ejemplo:

```
public class SelectDesconocido
{
   public static void main(String args[])
   {
   if (args.length == 0)
   {  System.out.println("Necesito un argumento.");
   }
```

JDBC. Metadatos de ResultSet

```
else
    {
      try
      {
       SQLServerDataSource ds = new SQLServerDataSource();
       ds.setIntegratedSecurity(true);
       ds.setServerName(“localhost");
       ds.setPortNumber(1433);
       ds.setDatabaseName("hotel");
       Connection con = ds.getConnection();
       Statement stmt = con.createStatement();
```

JDBC. Metadatos de ResultSet

```
ResultSet rset = stmt.executeQuery(args[0]);
SQLServerResultSetMetaData rsetMD = rset.getMetaData();

for(int i=1; i<=rsetMD.getColumnCount(); i++)
{
   System.out.print(rsetMD.getColumnName(i)+" ");
}
System.out.println("");
```

JDBC. Metadatos de ResultSet

```
while(rset.next())
{
    for(int i=1; i<=rsetMD.getColumnCount(); i++)
    {
       if (rsetMD.getColumnType(i) == Types.VARCHAR)
              System.out.print(rset.getString(i)+" ");
       else
              System.out.print(""+rset.getDouble(i)+" ");
       System.out.println("");
    }
}
```

JDBC. Metadatos de ResultSet

```
        rset.close();
        stmt.close();
        con.close();
       }
       catch(SQLException x)
       {
              x.printStackTrace();
       }
    }
  }
}
```

JDBC. Metadatos de la base de datos

- Los metadatos accesibles por la sesión actual pueden obtenerse a través de la conexión.
- Para ello se emplea el método `getMetaData()` que devuelve un objeto de tipo `SQLServerDatabaseMetadata` .
- El siguiente ejemplo muestra las tablas de las que es propietario el usuario actual.

```
SQLServerDatabaseMetadata dbaseMD = conn.getMetaData();
ResultSet rset = dbaseMD.getTables(“hotel”,null, null, null);
while(rset.next()){
    System.out.print(rset.getString("TABLE_CAT")+" ");
    System.out.print(rset.getString("TABLE_SCHEM")+" ");
    System.out.print(rset.getString("TABLE_NAME")+" ");
    System.out.println(rset.getString("TABLE_TYPE")+" ");
}
```

JDBC. Metadatos de la base de datos

- `getCatalogs`: Recupera los nombres del catálogo (nombre de las bases de datos) que están disponibles en el servidor conectado.
- `getFunctions`: Recupera una descripción de las funciones de usuario y del sistema.
- `getProcedures`: Recupera una descripción de los procedimientos almacenados que están disponibles en un modelo de nombre determinado de catálogo, esquema o procedimiento.
- `getTables`: Recupera una descripción de las tablas que están disponibles en el patrón de nombre determinado de catálogo, esquema o tabla.

SQLJ

- SQLJ supone una interesante forma estándar de acceder a una base de datos.
- SQLJ es un estándard ISO (ISO/IEC 9075-10) para embeber sentencias SQL en programas Java.
- Al contrario que JDBC, SQLJ no es una API sino una extensión del lenguaje.
- Los programas SQLJ deben ejecutarse a través de un preprocesador (el traductor SQLJ) antes de que puedan ser compilados.

SQLJ

- SQLJ tiene varias ventajas sobre JDBC:
  - Los programas SQLJ son más fáciles de escribir y de mantener. Además tienden a ser más cortos que los programas JDBC equivalentes.
  - Es más eficiente que JDBC dado que las sentencias SQL son analizadas y los caminos de acceso son optimizados en tiempo de compilación en lugar de en tiempo de ejecución.
  - Suministra mejor control de autorización: La Autorización puede ser concedida a los programas en lugar de a los usuarios.
  - Los problemas de rendimiento potenciales, tales como las consultas ineficientes debido a un mal camino de acceso, pueden ser identificados en tiempo de desarrollo.

SQLJ

- Hay varias desventajas:
  - Las sentencias SQL que se ejecutan deben ser estáticas.
  - SQLJ requiere un paso de preprocesamiento.
  - Muchos IDEs no proporcionan soporte SQLJ.
  - No hay soporte de SQLJ para la mayoría de frameworks de persistencia comunes

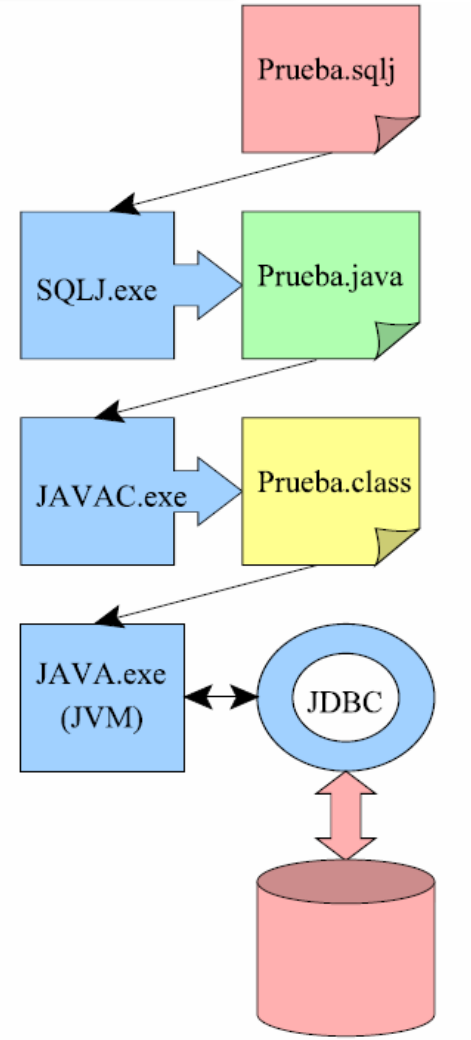
SQLJ

- El proceso de trabajo con la base de datos es muy parecido al de JDBC:
  - carga del driver
  - conexión
  - captura de excepciones
  - cierre de la conexión

- Tan sólo varía un poco la recuperación de datos y la asignación de valores en sentencias preparadas.

SQLJ

- Un fichero con extensión .sqlj es un fichero Java que tiene inmersas directivas SQLJ.
  El proceso para convertir un .sqlj en un .class es el siguiente:



Declaraciones SQLJ

- SQLJ permite dos tipos de declaraciones: declaraciones puras y de variables Java.
- Las declaraciones puras permiten declarar un `Iterator` (para recorrer el resultado de un `SELECT`) o bien la `Connection` por defecto con la que se van a hacer las operaciones (por si hay más de una conexión simultánea). Ejemplo.:

```
#sql public iterator EmpIter (int empno,
                              int enmae, int sal);
```

ó

```
#sql public context miConnCtxt;
```

Declaraciones SQLJ

- Tras la directiva `#sql` se coloca la sentencia SQL a ejecutar entre llaves. Esta sentencia puede contener expresiones Java precedidas por dos puntos. Ejemplo:

```
int hDeptno;
#sql {UPDATE emp SET sal=sal*1.5 WHERE deptno =
   :hDeptno};


float hNuevoSal;
#sql {UPDATE emp SET sal=:(1.5*hNuevoSal) WHERE
   deptno =5};
```

Declaraciones SQLJ

- También es posible ejecutar una función SQL y guardar el resultado ejecutando:

```
int hSal;
#sql hSal = {VALUES (funcion_SQL(parámetros))};
```

- Si nuestro programa realiza varias conexiones simultáneas, entonces hay que indicarle a las sentencias SQLJ cuál queremos utilizar.

SQLJ. Establecimiento de la conexión

- Para realizar una conexión, es posible crear un fichero connect.properties que contiene la información de conexión. Ejemplo:

```
sqlj.url=jdbc:oracle:thin:@localhost:1521:oracle
sqlj.user=alumno
sqlj.password=alumno
```

- Y la conexión se haría con:

```
Oracle.connect(getClass(),“connect.properties”);
```

- La conexión también puede hacerse de la forma:

```
Oracle.connect(“jdbc:oracle:thin:@localhost:1521:oracle”,
    “alumno”, “alumno”);
```

SQLJ. Establecimiento de la conexión

- Lo primero que hay que hacer es almacenar las distintas conexiones en forma de objetos `DefaultContext`.
- Para ello utilizaremos `getConnection()` para conectarnos a la base de datos.
- Desde Java se puede indicar la conexión por defecto ejecutando:

```
DefaultContext.setDefaultContext(contexto);
```

- En la propia directiva #sql se puede indicar el contexto de ejecución:

```
#sql contexto {sentencia SQL};
```

SQLJ. Ejemplo Insertar Registro en Tabla

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class Primer
{
public static void main(String[] args) throws SQLException {
if (args.length != 2) System.out.println("Faltan argumentos.");
else
{ Oracle.connect(Primer.class, "connect.properties");
  #sql {INSERT INTO EMP(EMPNO,ENAME) VALUES (:(args[0]),:(args[1]))};
  #sql {COMMIT};
  Oracle.close();
}}}
```

SQLJ. Ejemplo SELECT con resultado simple

```
Oracle.connect(SelectSimple.class, "connect.properties");
String ename;
double sal;
#sql {SELECT ename, sal INTO :ename, :sal
      FROM emp WHERE empno = :(args[0])};

System.out.println("Nombre: "+ename+". Salario: "+sal);
Oracle.close();
```

SQLJ. SELECT con resultado múltiple

- Cuando el SELECT devuelve cero o más de una fila, debe asignarse a un iterador.
- Existe una correspondencia unívoca entre cada tipo de iterador y cada tipo de tupla retornada por un SELECT.
- Un ejemplo de declaración de iterador sería:

```
#sql iterator EmpIter(int empno,String ename,double sal);
```

- que será capaz de recorrer un ResultSet con tres campos de los tipos especificados, por ese orden.
- Un iterador se asocia a una consulta de la forma:

```
EmpIter empRow = null;
#sql empRow = {SELECT empno, ename, sal FROM emp};
```

- produciéndose una correspondencia de campos por nombre.

SQLJ. SELECT con resultado múltiple

- Un iterador posee el método next() (con idéntico comportamiento a un ResultSet) y funciones con los nombres de los campos con los que se declaró. Ejemplo.:

```
while (empRow.next())
{
   System.out.print(empRow.empno()+"-");
   System.out.print(empRow.ename()+"-");
   System.out.println(empRow.sal()+"-");
}
```

- El iterador debe cerrarse con close():

```
empRow.close();
```

SQLJ. Ejemplo SELECT con resultado múltiple

```
Oracle.connect(SelectCompuesto.class,
    "connect.properties");
#sql iterator EmpIter(int empno,String ename,double sal);
EmpIter empRow = null;
#sql empRow = {SELECT empno, ename, sal FROM emp};
while (empRow.next()){
    System.out.print(empRow.empno() + " - " ) ;
    System.out.print( empRow.ename () + " - " ) ;
    System.out.println(empRow.sal()+"-");
}
empRow.close(); Oracle.close();
```

SQLJ. Control de transacciones

- En SQLJ el auto commit está desactivado por defecto.
- Para activarlo es necesario obtener el contexto por defecto y, a partir de él, obtener un objeto `Connection` al que hacerle un `setAutoCommit(true)`:

```
cntxt.getConnection().setAutoCommit(true);
```

- Estando el auto commit a false, si se cierra el contexto sin hacer un COMMIT, entonces se hace un ROLLBACK de la última transacción.