JDBC

Material original de Luis Vinuesa

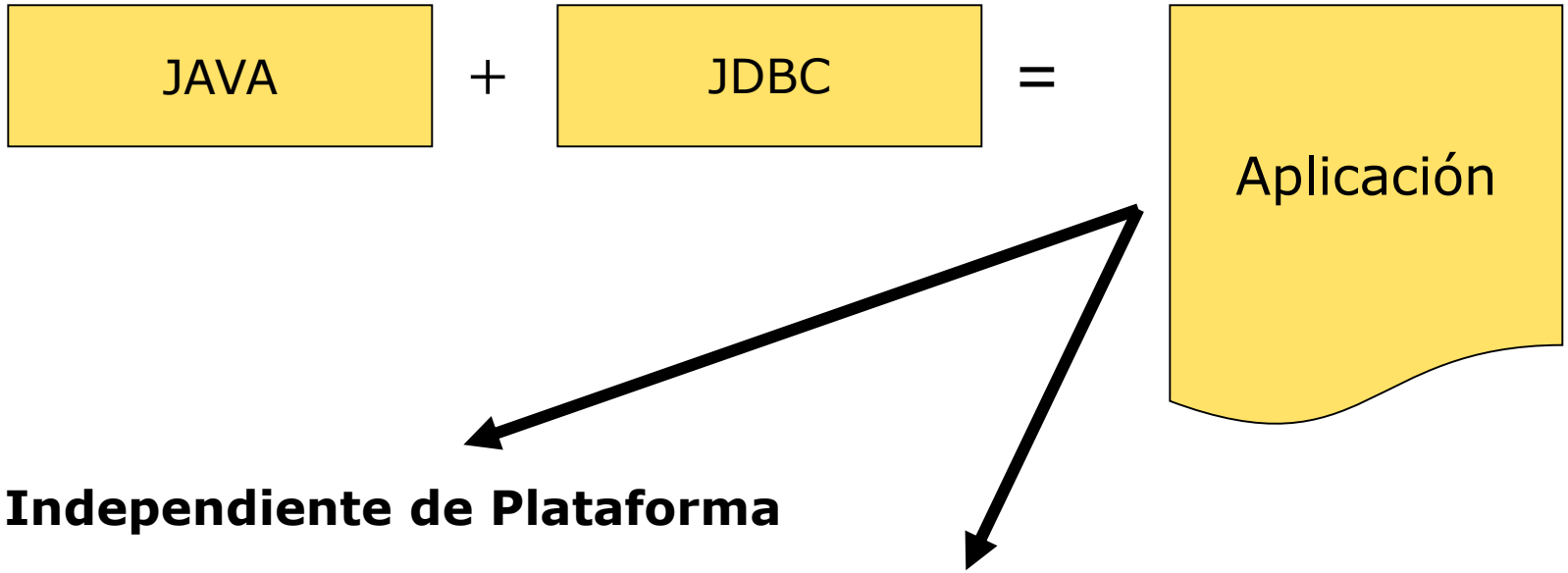
Contenido

- **Fundamentos de JDBC**
  - Qué es JDBC
  - Establecimiento de conexión
  - Ejecución de sentencias
  - Invocación a procedimientos almacenados
  - ResultSets y Cursores
  - Control de errores
- Otros tipos de conexión
- Pool de conexiones
- RowSet

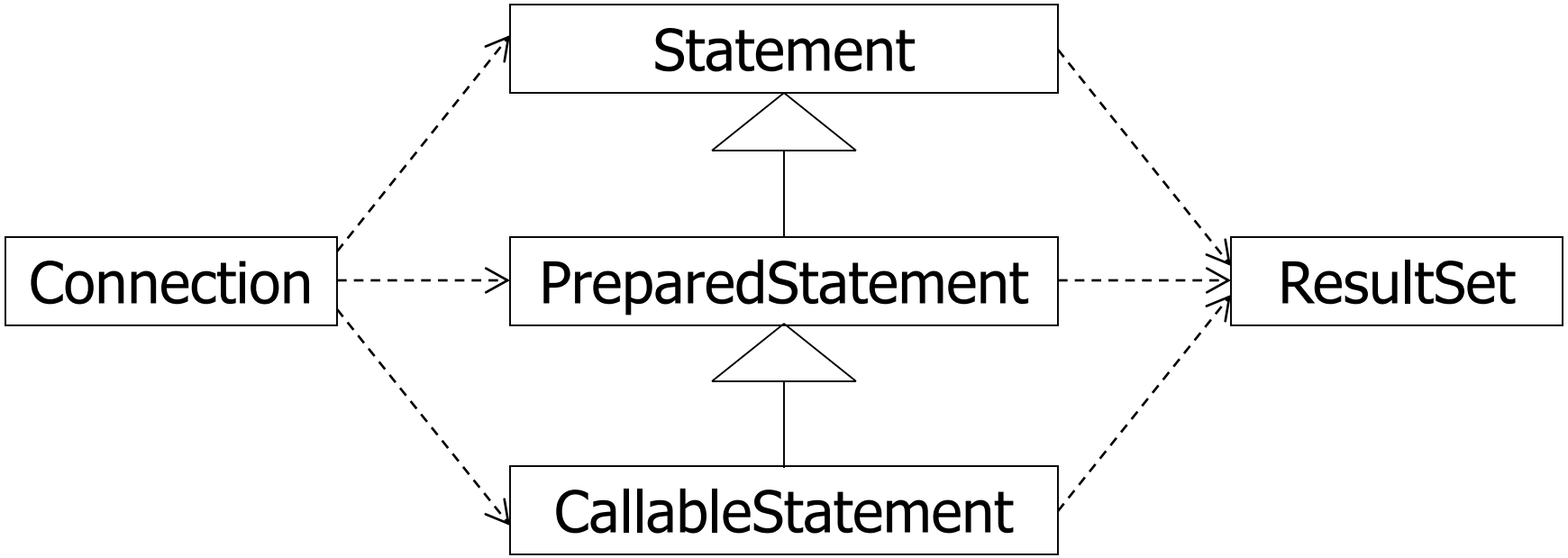
¿Qué es JDBC?

- JDBC es un conjunto de clases para la ejecución de sentencias SQL.
  - Ha sido desarrollado conjuntamente por JavaSoft, Sybase, Informix e IBM entre otros.
  - Permite manipular cualquier base de datos SQL. No es necesario un programa para manipular Oracle, otro para Sybase, etc. Un mismo programa puede manipular cualquier base de datos SQL.

Ventaja


JAVA
+
JDBC
=
Aplicación
Independiente de Plataforma
Independiente de la Base de Datos

Jerarquía de clases



Ejemplo JDBC

1. Establecer conexión
2. Realizar consulta
3. Cerrar todo

Control de errores

4

```
Connection con = null;
Statement stmt = null;
ResultSet rs = null;
try {
    con = DriverManager.getConnection(URL, USER, PASS);
    stmt = con.createStatement();
    rs = stmt.executeQuery("SELECT a, b FROM tabla");
    while (rs.next()) {
        int i = rs.getInt("a");
        String s = rs.getString("b");
        System.out.println( i + " -> " + s );
    }
}
catch (SQLException e){
    System.out.println("Error en consulta");
}
finally {
   if (rs != null){ try { rs.close(); } catch (SQLException e){}
   if (stmt != null){ try { stmt.close(); } catch (SQLException e){}
   if (con != null){ try { con.close(); } catch (SQLException e){}
}
```

1

2

3

1. Establecer conexión

- Para establecer la conexión con el SGBD son necesarios dos pasos:
  - 1) Cargar el driver
  - 2) Realizar la conexión

1. Establecer conexión – Cargar el Driver (I)

- 1) Es necesario conocer el nombre de las clases de los controladores JDBC utilizados por el fabricante
  - Ej1. COM.ibm.db2.jdbc.app.DB2Driver
  - Ej2. oracle.jdbc.driver.OracleDriver

- 2) Es necesario encontrar la librería donde se encuentra el controlador. Hay que indicar la ruta completa a ese controlador en la ruta de clases. Para ello, diversos mecanismos:
  - Lanzar los programas de bases de datos con el argumento de línea de comandos –classpath
  - Modificar la variable de entorno classpath
  - Copiar la librería de base de datos en el directorio jre/lib/ext.

1. Establecer conexión – Carga del Driver (II)

- 3) Antes de que el administrador de controladores pueda activar un driver, éste tiene que estar registrado. Una posibilidad para registrarlo es cargando su clase con el método forName (el cual provoca que se cargue la clase y se invoque a registerDriver)
  - Otra posibilidad es utilizar la opción -Djdbc.drivers al lanzar la ejecución de la JVM
  - Otra posibilidad es utilizar el método registerDriver de la clase DriverManager
  - Ejemplos:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("org.postgresql.Driver");
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

- **Desde JDBC 4.0 los drivers se registran automáticamente**

1. Establecer conexión - Realizar la conexión

- La conexión se abre mediante el método getConnection al que se le pasa un URL similar a:
  - **<protocolo>:<subprotocolo>:<parámetros específicos>**
    - Protocolo: siempre **jdbc**
    - Subprotocolo: nombre del driver
    - Parámetros: elementos tales como usuarios, contraseña, servidor, puerto, etc.
  - Ejemplos
    - **jdbc:oracle:thin:@<maquina>:1521:<instancia>**
    - **jdbc:hsqldb:file:nombre**
    - **jdbc:hsqldb:hsql://localhost/**
    - **jdbc:postgresql://localhost/testdb**
- En nuestro ejemplo:
  - private static String URL = "jdbc:oracle:thin:@156.35.94.99:1521:DESA";
  - private static String USER = "UOalumno";
  - private static String PASS = "PSWalumno";

```
con = DriverManager.getConnection(URL, USER, PASS);
```

2. Realizar consulta – Sentencia y ejecución

- Para poder ejecutar un comando SQL en primer lugar es necesario crear un objeto del tipo de sentencia que vayamos a utilizar:

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

- Especificar la sentencia SQL a ejecutar:

```
String sentencia = " Select * from libros ";
```

- Ejecutar la instrucción:

```
stat.executeQuery(sentencia);
```

- El método *executeQuery* sirve para lanzar (ejecutar) instrucciones Select del lenguaje SQL

Tipos de sentencias

- Hay 3 tipos:
  - **Statement**: para SQL (DML, DDL) sin parámetros
  - **PreparedStatement**: para SQL con parámetros o ejecuciones repetidas. Más eficiente y más seguro. **Usar siempre en vez de Statement** (para DML).
  - **CallableStatement**: para la invocación a procedimientos almacenados (lo vemos más adelante)
- Todos se obtienen a través de un elemento Connection

Statement - Ejecución

- executeQuery(<SQL>, ...)
  - Para ejecutar consultas: “SELECT ...”
  - Siempre devuelve un ResultSet
- executeUpdate(<SQL>, ...)
  - Ejecutar sentencias DDL y DML
  - Devuelve el nº de filas afectadas
- execute(<SQL>, ...)
  - Devuelve boolean indicando tipo resultado.

Statement - Batch

(implementación no estándar y no obligatoria)

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

stmt.addBatch("INSERT INTO emp VALUES …");
stmt.addBatch("INSERT INTO dep VALUES …");
stmt.addBatch("INSERT INTO emp_dept VALUES …");

int [] updateCounts = stmt.executeBatch();
```

PreparedStatement

- Sentencias SQL precompiladas
  - Mayor rendimiento si se van a ejecutar repetidas veces (en una aplicación real generalmente es lo que ocurre)
  - Ofrece más seguridad (p.ej: ante SQL Injection)

- Admite parámetros de entrada que vienen identificados por el símbolo ?

PreparedStatement - Parámetros

- Antes de ejecutar la sentencia es necesario darle valor a **todos** los parámetros (placeholders "**?**")
- En cada repetición se pueden cambiar los valores de los parámetros (se **deben** asignar valores)
- Para asignar el valor se utiliza el método set<tipo>(pos,valor), siendo el tipo compatible con el parámetro. Requiere dos argumentos
  - La posición del parámetro a asignar dentro de la sentencia
  - El valor a asignar al parámetro.

PreparedStatement - Parámetros

- Existen setters para todos los tipos básicos
  - setInt, setLong, setString,….
- La posición comienza por el **1**

```
...
consulta= "Select count(*) cuantos from bd.coches where
codcoche=?";
PreparedStatement ps = con.prepareStatement(consulta);
...
ps.setInt(1,i);
rs = ps.executeQuery();
...
```

PreparedStatement - Parámetros

- Si necesitamos asignar un valor nulo a un parámetro se usa el método setNull(pos,tipo), que recibe
  - Posición del parámetro
  - Tipo de dato SQL (no Java) del parámetro

```
pstmt.setNull(1,java.sql.Types.VARCHAR);
pstmt.setNull(2,java.sql.Types.NUMERIC);
pstmt.setNull(3,java.sql.Types.BLOB);
```

PreparedStatement - Tipos de datos

- El driver debe mapear:
  - Tipos java a tipos java.sql.Types.XXXX
  - java.sql.Types.XXXX a tipos nativos del DBMS
- En la documentación de Java y del fabricante del driver se especifican las equivalencias

PreparedStatement - Ejecución

- Puesto que PreparedStatement hereda de Statement las formas de ejecución son las mismas:
  - executeQuery(...)
  - executeUpdate(...)
  - execute(...)
- **NO** llevan SQL en la invocación, puesto que la sentencia ya ha sido fijada antes

```
PreparedStatement ps = con.prepareStatement(
"UPDATE coches SET modelo = ? WHERE cod = ?" );
ps.setString(1, "Ibiza");
ps.setInt(2, 25);
ps.executeUpdate(); //sin SQL
```

PreparedStatement - Resumen

- Especificar la sentencia SQL identificando las variables de servidor con **?**

```
String consulta1 = "SELECT Libros.precio, Libros.titulo "+
                   "FROM Libros, Editoriales "+
                   "WHERE Libros.codEditorial =
  Editoriales.CodEditorial and Editoriales.Nombre= ?";
```

- Crear el PreparedStatement

```
PreparedStatement psConsulta= conn.prepareStatement(consulta1);
```

- Asignar a cada variable su valor a través del método set.

```
psConsulta.setString(1,"McGrawHill"); //OJO: EMPIEZA EN 1
```

- Ejecutar la consulta

```
ResultSet rs = psConsulta.executeQuery();//OJO: SIN PARÁMETROS
```

CallableStatement

- Sentencias que llaman a procedimientos almacenados
  - Procedimientos o funciones
- Pueden devolver
  - ResultSet
  - Valores discretos en parámetros OUT e INOUT
- Sintaxis ODBC para invocar a proc's.

CallableStatement – Sintaxis

{[? =] call procedure_name[(?, ?, ...)]}

“{call procedure}”
“{call procedure(?, ?)}”
“{? = call function}”
“{? = call function(?, ?)}”

CallableStatement - Parámetros

- Puede haber parámetros de entrada (IN), salida (OUT) y de entrada y salida (IN OUT).
- Además las funciones devuelven valores.

CallableStatement - Parámetros

- Parámetros de entrada
  - Igual que en el caso de los PreparedStatement
  - Si se necesita indicar valor nulo se hace igual que en PreparedStatement (setNull)

```
CallableStatement cs = con.prepareCall("{call MUESTRA_EDITORIALES (?,?)}");
  cs.setString(1,"xyz");
  cs.setString(2,"ab");
  cs.executeQuery();
```

CallableStatement - Parámetros

- Parámetros de salida
  - Se referencian también por posición
  - Deben ser registrados antes de la ejecución (registerOutParameter(pos,tipo)..)

```
CallableStatement cs = con.prepareCall("{call CALCULA_MEDIA (?)}");
 cs.registerOutParameter(1,java.sql.Types.FLOAT);
 cs.executeQuery();
 float media = cs.getFloat(1); //se obtiene del cs
```

CallableStatement - Parámetros

- Parámetros de salida nulos
  - Se detectan con el método callableStatement.wasNull()
  - Debe ser llamado después del getter

```
byte x = cstmt.getByte(1);
if (cstmt.wasNull())
{ . . .
```

  - Puede que el manipulador del resultado sepa tratar con valores null.

CallableStatement - Parámetros

- Parámetros de entrada-salida
  - Se referencian también por posición
  - Combina los métodos anteriores

  Método set<TIPO>(<pos>,<valor>)

  registerOutParameter(…)

  Execute[<modo>]()

  Método get<TIPO> (<pos>)

CallableStatement - Parámetros

- Ejemplo IN OUT

```
CallableStatement cs = con.prepareCall("{call
ACTUALIZA_PRECIOS (?)}");
cs.setByte(1,(byte)25);
cs.registerOutParameter(1,
     java.sql.Types.TINYINT);
cs.executeUpdate();
byte x = cs.getByte(1);
```

CallableStatement - funciones

- Ejemplo de invocación a una función con un parámetro de entrada

```
CallableStatement cs = con.prepareCall("{? = call
CALCULA_MEDIA (?)}");

cs.registerOutParameter(1,java.sql.Types.FLOAT);
cs.setString(2,mdni);

cs.executeQuery();
float media = cs.getFloat(1);
```

CallableStatement – Ejecución

- Se ejecuta de la misma manera que PreparedStatement
- Recordatorio:
  - PreparedStatement extends Statement
  - CallableStatement extends PreparedStatement

CallableStatement - Resumen

- Crear un objeto CallableStatement, que contiene la llamada al procedimiento almacenado y no el procedimiento en sí mismo

  CallableStatement cs = con.prepareCall("{call MUESTRA_EDITORIALES}");

  ResultSet rs= cs.executeQuery(); o bien cs.executeUpdate();

- Puede tener parámetros (IN/OUT) y devolver resultados (función)

  CallableStatement cs = con.prepareCall("{? = call CALCULA_MEDIA (?)}");

  cs.registerOutParameter(1,java.sql.Types.FLOAT);

  cs.setString(2,mdni);

  cs.executeQuery();

  float media = cs.getFloat(1); //OJO: se obtiene del propio CallableStatement

2. Realizar consulta – Recoger resultados

- Cuando se ejecuta una consulta SQL, lo importante es el resultado.
- El objeto *executeQuery* devuelve un objeto de tipo *ResultSet* que se puede emplear para procesar las filas del resultado

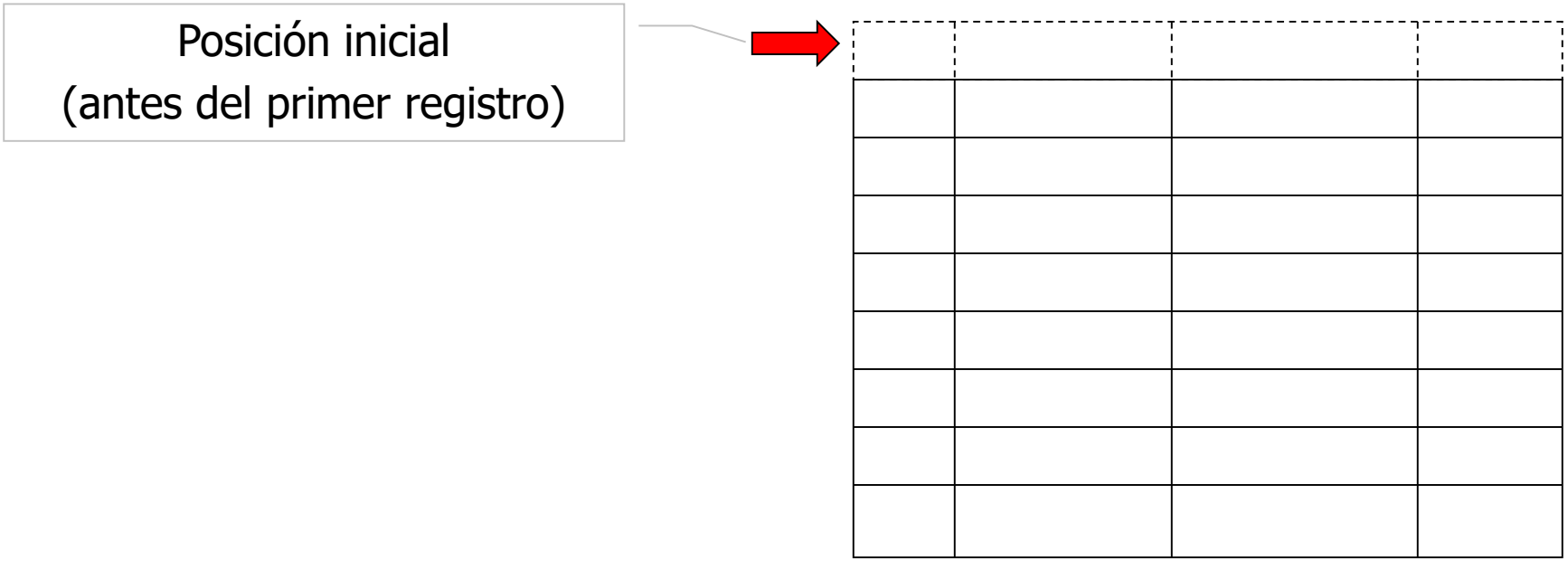
```
ResultSet rs = stat.executeQuery ("Select * from libros");
```

- Para analizar el conjunto de resultados:

```
while (rs.next()){
      String isbn = rs.getString(1);
      float precio = rs.getDouble("Precio"); }
```

ResultSet

- Conjunto de filas + cursor
- Devuelto en todos los métodos executeQuery(...)
- También lo pueden devolver los métodos execute(...)



ResultSet - Ejemplo

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery( "SELECT a, b, c FROM tabla" );
while (rs.next()) {
    int i = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");

    System.out.println(i+" "+s+" "+f);
}
```

Cursores

- Indican la fila activa del ResultSet
- Pueden ser:
  - Sólo hacia delante
  - Bidireccionales
- Por defecto sólo FORWARD (menos recursos)
- Para crearlos bidireccionales:
  - Indicación expresa en Connection al crear la sentencia

Movimientos del Cursor

- rs.first()
- rs.beforeFirst()
- rs.last()
- rs.afterLast()
- rs.next()
- rs.previous()
- rs.absolute(5)//a la quinta fila
- rs.relative(-3)//3 filas antes

Control del Cursor

- rs.isBeforeFirst()
- rs.isAfterLast()
- rs.isFirst()
- rs.isLast()

Datos de columnas

- Métodos getter:
  - rs.get<TIPO>(posición | nombre);
  - Hay que indicar la posición o el nombre de la columna
    - Mejor opción el nombre, ya que si cambia la consulta o la definición de la tabla sigue funcionando

```
String s = rs.getString(2);
String s = rs.getString("codigo");
```

Tipos de ResultSet

- Según:
  - Movimiento del cursor
  - Si permiten ver cambios hechos por otros usuarios mientras está abierto
- 3 tipos:
  - TYPE_FORWARD_ONLY
  - TYPE_SCROLL_INSENSITIVE
  - TYPE_SCROLL_SENSITIVE
- Para conseguir el scroll se utiliza memoria en el cliente (JVM). Hay que usarlo con cuidado.
- La implementación depende del fabricante, no todos lo soportan

Tipos de Concurrencia

- Forma en la que varios usuarios trabajan sobre los mismos datos:
  - CONCUR_READ_ONLY
    - Impone bloqueo de lectura
  - CONCUR_UPDATABLE
    - Impone bloqueo de escritura
    - Restringe mucho la concurrencia, se debe administrar con mucha cautela

Tipos de Concurrencia

- La implementación depende del driver, puede tener limitaciones (y no todos lo soportan)
- Oracle
  - Sólo bloquea la fila cuando se ejecuta updateRow, no al crear el rs (si se necesita bloquear desde el principio se puede usar SELECT … FOR UPDATE)
  - En la select sólo puede haber una tabla (no una join)
  - En la selección sólo puede haber columnas, no cálculos ni agregadas (SUM, MAX…) (sí se pueden usar en el WHERE, etc)

Retenibilidad (Holdability)

- Los ResultSet podrían permanecer en memoria del cliente después de terminar la transacción que los creó.
- Dos modos:
  - ResultSet.HOLD_CURSORS_OVER_COMMIT
  - ResultSet.CLOSE_CURSORS_AT_COMMIT

3. Cierre de la conexión

```
        ...
    }
    finally {
        if (con != null){
            try {
                con.close();
            }
            catch (SQLException e){
                // Se suele dejar vacío en este caso particular
            }
        }
    }
```

Se suele compactar en esto:

```
        ...
    }
    finally {
        if (con != null) try { con.close(); } catch (SQLException e){}
    }
```

4. Control de errores

- Cuando ocurre un error se lanza una excepción java.
- Las excepciones que lanzan las operaciones JDBC son de tipo SQLException

```
try{
        ... Código JDBC
}catch(SQLException e){
        ... Control del error
}
```

4. Control de errores - SQLException

- Principales métodos de SQLException
  - **String getSQLState()**, identifica el error de acuerdo a X/Open
  - **int getErrorCode()**, obtiene el código de excepción específico del fabricante
  - **String getMessage()**, obtiene una cadena que describe la excepción
  - **SQLException getNextException()**, obtiene la excepción encadenada de ésta

```
try{…

} catch (SQLException ex){
                    System.out.println(" SQLException recogida: ");
                    while (ex!=null){
                              System.out.println("Mensaje: "+ex.getMessage());
                              System.out.println("SQLState: "+ex.getSQLState());
                              System.out.println("ErrorCode: "+ex.getErrorCode());
                              ex=ex.getNextException();
                              System.out.println(" ");
                    }
}
```

4. Control de errores –SQLWarning (I)

- Los objetos SQLWarning son una subclase de SQLException que tratan los avisos en el acceso a las bases de datos
- A diferencia de las excepciones los warnings **no interrumpen** la ejecución de una aplicación, es **responsabilidad del programador** consultar para comprobar si ha ocurrido un warning
- Ejemplos de warnings:
  - Realizar un SUM que contiene valores null
  - Intentar fijar un nivel de aislamiento que no es soportado por el sistema
  - Fijar un tipo de ResultSet no compatible con la sentencia: updatable con una select que contiene una join de varias tablas.
- Los warnings pueden ser reportados sobre objetos Connection, Statement (y las clases que la extienden como PreparedStatement y CallableStatement) o un ResultSet.

4. Control de errores –SQLWarning (II)

- Existe un método **getWarnings** (en Connection, Statement y ResultSet) que puede/debe ser invocado con el objetivo de obtener el primer warning producido
- Si getWarnings devuelve un warning se puede invocar al método **getNextWarning** para recoger cualquier warning adicional que se haya producido

```
while (rs.next()){
        …
SQLWarning war =rs.getWarnings();
While (war!=null){
                System.out.println("Mensaje: "war.getMessage());
                System.out.println("SQLState: "+war.getSQLState());
                System.out.println("ErrorCode: "+war.getErrorCode());
                war=war.getNextWarning();
                System.out.println(" ");
                }
}
```

4. Control de errores – finally

- Es **muy importante** cerrar los recursos que vamos abriendo con **finally**

```
try{
        ... Código JDBC
} finally {
        ... Acción con o sin error
        Cerrar siempre RS, Stmt y Connection
}
```

```
finally {
   if (rs != null){ try { rs.close(); } catch (SQLException e){}
   if (stmt != null){ try { stmt.close(); } catch (SQLException e){}
   if (con != null){ try { con.close(); } catch (SQLException e){}
}
```

Contenido

- Fundamentos de JDBC
  - Qué es JDBC
  - Establecimiento de conexión
  - Ejecución de sentencias
  - Invocación a procedimientos almacenados
  - ResultSets y Cursores
  - Control de errores
- **Otros tipos de conexión**
- **Pool de conexiones**
- RowSet

Tipos de Conexión

- Para obtener una conexión (Connection) hay 3 alternativas
  - Usar el DriverManager
  - Usar un DataSource
  - Instanciar el driver directamente
- En cualquier caso se obtiene un objeto de tipo Connection y a partir de él se crean Statement, PreparedStatement, Callable…

Conexión - DriverManager

- Registra todos los Tipos de DriversJDBC
  - El driver a usar debe ser registrado previamente (mediante forName, registerDriver, etc). (no es necesario >=JDBC 4.0)
- Al crear la conexión se le indica el URL del driver
  - **<protocolo>:<subprotocolo>:<parámetros específicos>**
- El DM localiza el driver adecuado buscando el URL entre los que tiene registrados y establece la conexión (Si no hay errores)

Conexión - DriverManager

```
DriverManager.registerDriver(
  new oracle.jdbc.driver.OracleDriver()
);
conn = DriverManager.getConnection(
  "jdbc:oracle:thin:@156.35.94.99:22:DESA"
  , "scott"
  , "tiger"
);
```

Conexión - DataSource

- Desde JDBC 2.0. Representa cualquier fuente de datos (Como si fuese un alias o DSN ODBC).
- Muchas similitudes con DriverManager
- Importantes diferencias:
  - Se registra en un árbol JNDI (también se puede usar sin registrar)
    - Independencia del programa
  - Permite gestionar pools de conexiones
    - Aumenta la eficiencia si se abren y cierran muchas conexiones
  - Permite transacciones distribuidas

Conexión – Registro JNDI

- El DataSource se registra en el árbol JNDI bajo una clave indicando la conexión, el driver y sus particularidades para un DBMS
  - Lo suele hacer un administrador del sistema/framework (Tomcat..)
- El programa pregunta por la clave y recibe el DataSource
- Es un nivel de indirección que independiza el programa del SGBD.
- Si cambia el SGBD, se cambian las propiedades en el JNDI, y el código del programa no se necesita modificar.

Conexión - DataSource

- dataSourceName
- databaseName
- description
- networkProtocol
- user
- password
- serverName
- port…

- Crear un DataSource
- Especificar esto
- Registrar en JNDI bajo una clave ej: “miBD”
- Programa busca la clave
- Pide una conexión

Conexión - DataSource

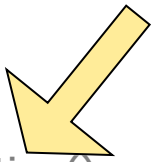
```
...
Context ctx = new InitialContext();
DataSource ds =
(DataSource)ctx.lookup("jdbc/distCoffeesDB");
Connection con = ds.getConnection();
...
```

Conexión - DataSource

```
...
OracleDataSource ods = new OracleDataSource();

ods.setDriverType("oci");
ods.setServerName("dlsun999");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("PROD");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");

Connection conn = ods.getConnection();
...
```



```
Context ctx = new InitialContext();
ctx.bind("jdbc/sampledb", ods);
OracleDataSource odsconn =
(OracleDataSource)ctx.lookup("jdbc/sampledb");
Connection conn = odsconn.getConnection();
```

Pool de conexiones (I)

- Abrir (y mantener) conexiones es costoso en tiempo y recursos.
  - A partir del JDBC 2.0 se soportan los pool de conexiones.
  - Para poder hacer uso de un pool es necesario obtenerlo a través de un **DataSource**.
- Puede venir implementado en el driver del fabricante o puede utilizarse un pool de conexiones genérico (válido con distintos drivers)

Pool de conexiones (II)

- Hay que distinguir entre
  - PooledConnection (Interfaz javax.sql.PooledConnection). Conexión física que puede ser "reciclada"
  - Pool de conexiones (implementado por algún fabricante o desarrollador, no definición standard)
    - OracleConnectionCacheManager (para/de Oracle)
    - c3p0
    - BoneCP
    - Jakarta DBCP

Pool de conexiones - Ejemplo

```
DataSource unpooled = DataSources.unpooledDataSource(
    "jdbc:oracle:thin:@156.35.94.99:22:DESA",
    "user",
    "password"
);

Map overrides = new HashMap();
overrides.put("maxStatements", "200");
overrides.put("maxPoolSize", new Integer(50));
overrides.put("minPoolSize","6");

DataSource pooled = DataSources.pooledDataSource( unpooled, overrides );

con = pooled.getConnection();
```

Contenido

- Fundamentos de JDBC
  - Qué es JDBC
  - Establecimiento de conexión
  - Ejecución de sentencias
  - Invocación a procedimientos almacenados
  - ResultSets y Cursores
  - Control de errores
- Otros tipos de conexión
- Pool de conexiones
- **RowSet**

RowSet

- Es un objeto que encapsula un conjunto de filas provenientes de un ResultSet o de una fuente de datos tabular.
  - Es una copia en memoria del resultado de la consulta
  - Se puede desconectar de la base de datos (no necesita cerrarse)
- Su interfaz deriva de la de ResultSet→se puede hacer lo mismo que con los ResultSet
- Es un componente JavaBean → se puede serializar
- No todos los fabricantes soportan que sus ResultSet sean "scrollables" o "updatables"→Un RowSet lo es por defecto (obligatoriamente)

RowSet

- Se introduce en JDBC 2.0 como paquete opcional.
- Se estandariza mediante JDBC RowSet Implementations Specification (JSR-114), el cual pasa a ser no opcional desde la versión Java SE 5.0, ampliando su API en la versión Java SE 6.0 y más en la Java SE 7.0

RowSet

- Al ser JavaBeans tienen
  - Propiedades. Con sus setters y getters
  - Mecanismo de notificación JavaBeans
- Para las notificaciones se basa en un *Listener* que responde ante 3 eventos:
  - Movimiento del cursor
  - Modificación, borrado o inserción de una fila
  - Un cambio en el contenido del RowSet completo (por ejemplo al hacer select)

RowSet - Tipos

- Existen 5 implementaciones de referencia de RowSet (son interfaces estandarizadas).
  - JdbcRowSet
  - CachedRowSet
  - WebRowSet
  - JoinRowSet
  - FilteredRowSet

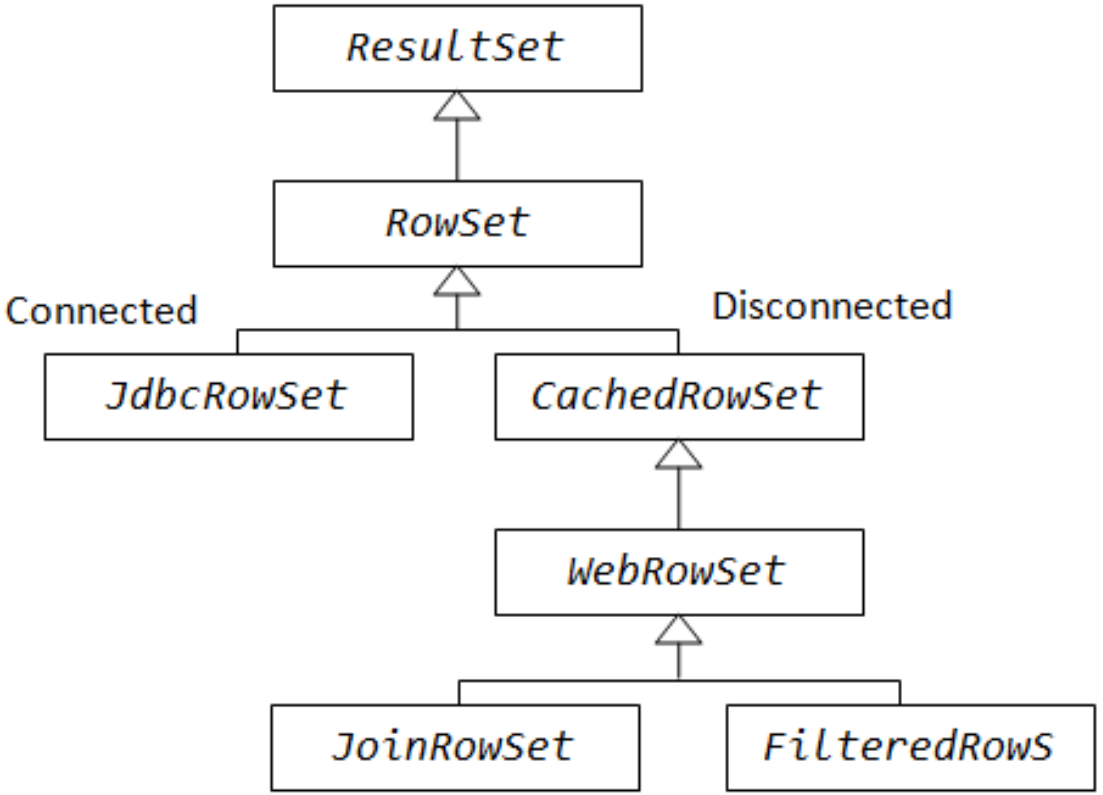
RowSet - Conexión

- Un RowSet puede ser de tipo **conectado** (JdbcRowSet) o **desconectado** (los demás).
- Si es conectado mantiene una conexión con el SGBD permanentemente (los cambios en el RowSet se reflejan en la tabla)

RowSet - Conexión

- Si es desconectado
  - Se conecta para recibir los datos (del SGBD)
  - Se desconecta y permite trabajar
  - Se conecta para enviar los datos (al SGBD)
    - Verifica los conflictos que se puedan haber producido y los resuelve
- Debido a esta forma de trabajar
  - Es ligero
  - Es serializable
- Se pueden transmitir por la red a distintos dispositivos, aplicaciones, etc

RowSet - Jerarquía



RowSet - Funcionalidad

- **JdbcRowSet** ofrece la misma funcionalidad de ResultSet garantizando que puede ser *scrollable y updatable* (aunque el ResultSet no lo sea)
- **CachedRowSet** ofrece (Además de la de JdbcRowSet) la funcionalidad de:
  - Manipular datos (y modificarlos) mientras está desconectado
  - Reconectarse para escribir los cambios al SGBD
  - Verificar posibles conflictos y solucionarlos

RowSet - Funcionalidad

- **WebRowSet** ofrece (Además de la de CachedRowSet) la funcionalidad de:
  - Escribirse como un documento XML
  - Leer un documento XML que lo describa
- **JoinRowSet** ofrece (Además de la de WebRowSet) la funcionalidad de:
  - Realizar el equivalente a una JOIN SQL sin tener que conectarse a la base de datos
- **FilteredRowSet** ofrece (Además de la de WebRowSet) la funcionalidad de:
  - Aplicar filtros a los datos sin tener que conectarse a la base de datos (WHERE). Sólo los datos seleccionados son visibles

RowSet – JdbcRowSet

- Un JdbcRowSet se puede crear mediante
  - La implementación del constructor de referencia (JdbcRowSetImpl())que recibe un ResultSet
  - La implementación del constructor de referencia que recibe una Connection
  - La implementación del constructor de referencia por defecto (sin parámetros)
  - Usando una instancia de RowSetFactory creada a partir de RowSetProvider (Java SE 7.0)
  - Mediante el constructor que ofrezca el driverJDBC que se use. Las implementaciones pueden variar de la de referencia, cambiando nombres, constructores…ej: OracleJDBCRowSet

RowSet - JdbcRowSet

- Cuando un JdbcRowSet se crea a partir de un ResultSet funciona de la misma forma respecto a *scrollable y updatable*. Si el ResultSet no lo es entonces el JdbcRowSet tampoco lo es.
- En el resto de casos estas opciones están activadas por defecto
  - ResultSet.TYPE_SCROLL_INSENSITIVE
  - ResultSet.CONCUR_UPDATABLE
  - Connection.TRANSACTION_READ_COMMITTED

RowsSet - JdbcRowSet

- Respecto al autocommit
  - Existe el método setAutoCommit(boolean)
  - Existen los métodos commit y rollback
- Por defecto activado
- Si se crea a partir de una conexión coge el valor que tenga en la conexión

RowsSet - JdbcRowSet

- Al igual que en el ResultSet las modificaciones, inserciones o eliminaciones de filas se ven afectadas por el modo de AutoCommitMode
- Si está activado al invocar al método (updateRow…) se realiza efectivamente la modificación y el commit
- Si no está activado se realiza la operación en la base de datos (se hace Post), pero no se realiza el commit hasta que lo hace explícitamente el programa dando lugar a bloqueos de datos (por ejemplo en un update)

RowSet – JdbcRowSet Creación

Ejemplo1 (A través de ResultSet)

```
stmt = con.createStatement(
           ResultSet.TYPE_SCROLL_SENSITIVE,
           ResultSet.CONCUR_UPDATABLE);
rs = stmt.executeQuery("select * from COCHES");
jdbcRs = new JdbcRowSetImpl(rs);//se crea y se rellena
```

Ejemplo2 (A través de Connection)

```
jdbcRs = new JdbcRowSetImpl(con);//se crea
jdbcRs.setCommand("select * from COCHES where cod=?”);
jdbcRs.setInt(1, 5000);
jdbcRs.execute();//en este momento se rellena
```

RowSet – JdbcRowSet Creación

```
….
jdbcRs = new JdbcRowSetImpl();//Se crea
jdbcRs.setCommand("select * from COCHES");
//para conectarse se puede usar url o un DataSource
//se conecta mediante un DataSource – tiene que existir
//jdbcRs.setDataSourceName(“PRUEBA");
//se especifica una url.
jdbcRs.setUrl("jdbc:myDriver:myAttribute");
jdbcRs.setUsername(username);
jdbcRs.setPassword(password);
jdbcRs.execute();//se rellena de datos
…
```

RowSet – JdbcRowSet Modificación datos

```
…
jdbcRS.execute();
jdbcRS.addRowSetListener(new ExampleListener());
while (jdbcRS.next()) {
System.out.println("cod=" + jdbcRS.getString(1));
}
jdbcRS.absolute(3);//se sitúa en la posición 3
jdbcRS.updateInt(1, x);//modifica la primera columna
con el valor de x
jdbcRS.updateRow();//lo hace efectivo – post o commit
jdbcRS.commit();//dependiendo del modo de AutoCommit
```

RowSet – JdbcRowSet Listener

```
jdbcRS.execute();
jdbcRS.addRowSetListener(new ExampleListener());
..
class ExampleListener implements RowSetListener {
  public void cursorMoved(RowSetEvent event) {
    System.out.println("se movió el cursor" +
event.toString());}
  public void rowChanged(RowSetEvent event) {
System.out.println(event.toString());}
  public void rowSetChanged(RowSetEvent event) {
System.out.println(event.toString());}
}
```

RowSet - JdbcRowSet

- Presenta las mismas limitaciones que ResultSet
  - Para que pueda ser *updatable* la select no puede tener JOIN
  - Tampoco puede ser SELECT *
  - Sólo bloquea la fila cuando se ejecuta updateRow, no al crear el rs (si se necesita bloquear desde el principio se puede usar SELECT … FOR UPDATE)

RowSet - CachedRowSet

- La principal característica es que puede operar sin estar conectado a su fuente de datos→desconectado. Almacena sus datos en memoria (caché) del cliente
- La fuente de datos no tiene por qué ser una base de datos relacional. Puede ser cualquier fuente de datos que los almacene de una forma tabular (una hoja de cálculo, un fichero plano…)
- La implementación por defecto obtiene los datos de una base de datos relacional

RowSet - CachedRowSet

- Un CachedRowSet se puede crear mediante
  - El constructor por defecto
  - Usando una instancia de RowSetFactory creada a partir de RowSetProvider (Java SE 7.0)
  - Mediante el constructor que ofrezca el driverJDBC que se use. Las implementaciones pueden variar de la de referencia, cambiando nombres, constructores…ej: OracleCachedCRowSet

RowSet - CachedRowSet

- Cuando se crea tiene las mismas opciones por defecto que un JdbcRowSet
- El sistema de notificaciones es idéntico al de los JdbcRowSet (mediante Listener)
- [Al estar desconectado es necesario indicarle qué columnas van a actuar de clave (key)] (en teoría)

```
int [] keys = {1};
crs.setKeyColumns(keys);
```

RowSet – CachedRowSet Modificación de datos

- Las modificaciones de datos (borrados, inserciones y modificaciones) se realizan de igual forma que en JdbcRowSet pero al estar desconectado hay que invocar al método **acceptChanges()** para que se conecte y realice las modificaciones
- Pueden existir conflictos y en ese caso salta una excepción SyncProviderException y mediante SyncResolver se pueden intentar resolver

RowSet - WebRowSet

- Hereda de CachedRowSet, por lo que presenta toda su funcionalidad.
- Además permite guardar su información en un fichero XML
  - writeXml(…)
- Y cargar su contenido de un fichero XML
  - readXml(…)

RowSet - JoinRowSet

- Hereda de WebRowSet, por lo que presenta toda su funcionalidad.
- Un JoinRowSet permite realizar JOIN entre distintos RowSet mientras no están conectados a la fuente de datos.
- Los RowSet se añaden mediante el método addRowSet(…)
- Se pueden añadir tantos como se desee

RowSet - JoinRowSet

- Existen varios tipos de Join
  - CROSS_JOIN
  - FULL_JOIN
  - INNER_JOIN – opción por defecto
  - LEFT_OUTER_JOIN
  - RIGHT_OUTER_JOIN
- Para que se pueda hacer la Join es necesario indicar qué columna(s) tiene(n) que igualarse. Esto se hace mediante
  - `setMatchColumn(int|int[]|String|String[])` de los RowSet a añadir(perteneciente a interfaz Joinable)
  - `addRowSet(RowSet, int|int[]|String|String[])`

RowSet – JoinRowSet Ejemplo

```
RS = new CachedRowSetImpl();…..  //se fija url, username…
              RS.setCommand("SELECT cifm,nombrem from bd.marcas");
              RS.execute();
              RS.setMatchColumn("CIFM");

RS2 = new CachedRowSetImpl();…… //se fija url, username…
    RS2.setCommand("SELECT codcoche,cifm from bd.marco");
    RS2.execute();
    RS2.setMatchColumn("CIFM");

JS=new JoinRowSetImpl();
JS.addRowSet(RS);
JS.addRowSet(RS2);

    while (JS.next()) {
                System.out.println("cifm=" + JS.getString("CIFM"));
                System.out.println("nombrem=" + JS.getString("NOMBREM"));
                System.out.println("codcoche=" + JS.getString("CODCOCHE"));
              }
```

RowSet – JoinRowSet Ejemplo

```
//continuación para añadir un tercer RowSet
 RS3 = new CachedRowSetImpl(); ….. //se fija url, username…
            RS3.setCommand("SELECT codcoche,nombrech FROM bd.coches");
            RS3.execute();
            RS3.setMatchColumn("CODCOCHE")
//Fijo la columna(s) del JoinRowSet→del Join ya hecho
JS.setMatchColumn("CODCOCHE");//del JoinRowSet
JS.addRowSet(RS3);

            while (JS.next()) {
              System.out.println("cifm=" + JS.getString("CIFM"));
              System.out.println("nombrem=" + JS.getString("NOMBREM"));
              System.out.println("codcoche=" + JS.getString("CODCOCHE"));
              System.out.println("nombrech=" + JS.getString("NOMBRECH"));
            }
```

RowSet - FilteredRowSet

- Permite aplicar filtros a un RowSet. Se pueden aplicar tantos como se desee (uno tras otro)
- Sólo son visibles las filas que cumplan el filtro.

RowSet - FilteredRowSet

- Para aplicar un filtro se invoca a
  - setFilter(Predicate); (Predicate es una interfaz)
  - Si se quiere aplicar un segundo (tercer, etc) filtro se vuelve a invocar
- Para eliminar los filtros se invoca a
  - setFilter(null);

RowSet - FilteredRowSet

```
FilteredRowSet frs = new FilteredRowSetImpl();
frs.setCommand("SELECT * FROM COFFEE_HOUSES");
….url, usuario….
frs.execute();//se rellena con datos
//Rango es una clase que implementa Predicate
//es equivalente a A<=columna<=H
Rango rango = new Rango("A", "H", "columna");

frs.setFilter(rango);//aplicamos el filtro

frs.next() // solo las filas que en columna tengan
//valores entre A y H se mostrarán
```

RowsSet - FilteredRowSet

- A la hora de modificar o insertar filas se puede hacer siempre que la fila resultante verifique el filtro (o los filtros) efectivos en ese momento
- A la hora de eliminar filas se puede hacer siempre que la fila a eliminar sea visible en ese momento