JDBC.

Víctor Herrero Cazurro

Febrero 2012

JDBC

1. **Introducción.**
2. **API.**
3. **Conexión.**
4. **Crear tablas.**
5. **Insertar registros.**
6. **Modificar registros.**
7. **Eliminar registros.**
8. **Consultar tablas.**
9. **ResultSet.**
10. **Metadatos.**
11. **PreparedStatement.**
12. **Procedimientos almacenados.**
13. **Transacciones.**
14. **Ampliación ResultSet.**

JDBC: Introducción

- **JDBC** proporciona a Java un mecanismo sencillo y potente de acceder a bases de datos relacionales.
- El paquete **java.sql** que se incluye desde la versión 1.2 y siguientes del API de Java contiene el API **JDBC**.
- Uno de los mayores problemas con las bases de datos es la rivalidad que hay entre los fabricantes. Esto implica que con cada base de datos de dialoga de forma distinta.

JDBC: Introducción

- Para los desarrolladores, lo ideal es disponer de un **único mecanismo de acceso** a cualquier base de datos relacional, de tal forma que los cambios en la aplicación para cambiar de base de datos sean mínimos.
- Esto lo resolvió Microsoft con su estándar **ODBC** (Open Data Base Connectivity): un controlador universal que permite acceder a muchas bases de datos de un modo estándar, sin tener que hacer cambios de código si se cambia de base de datos.
- El inconveniente de **ODBC** es que sólo está disponible en la plataforma Windows, por lo que Sun Microsystems desarrolló otro estándar, el **JDBC**.

JDBC: Introducción

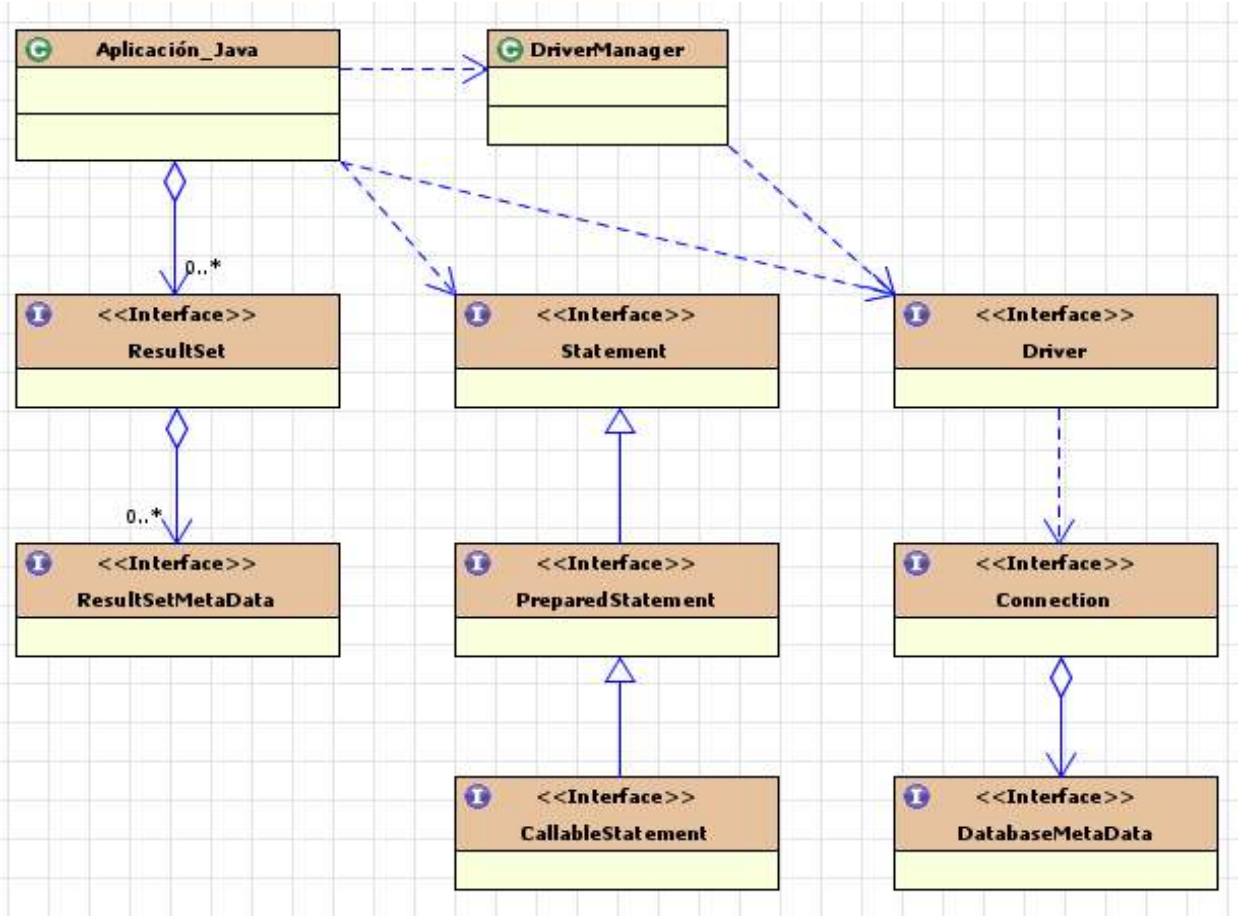
- Para interactuar con una base de datos, hay que establecer una conexión con el gestor e identificarte ante él.
- La identificación, es necesaria para que el gestor sepa quién intenta acceder a los datos. Un mecanismo de seguridad hace posible que el acceso a cierta información esté restringido, y que por ejemplo puedas consultar pero no modificar datos de ciertas tablas.
- Para establecer la conexión, se emplea el API **JDBC**, pero necesitaremos una implementación concreta para la base de datos a la que nos conectemos, esta implementación se conoce como controlador o **driver**.

JDBC: Introducción

- Estos **driver**, son desarrollados por los fabricantes de base de datos, y suministran la implementación de las interfaces del API **JDBC** que permiten interactuar con el servidor de base de datos a una aplicación Java.

JDBC: API

- Estructura y relaciones de clases e interfaces fundamentales en el paquete **java.sql**.



JDBC: API

- Relación y dependencias entre las clases del API **JDBC**.

Clase/Interfaz	Utilidad	Métodos	Cometido
DriverManager	Se encarga de leer todos los drivers	**getConnection(String)**	Obtiene un objeto Connection a partir de la "cadena de conexión".
Connection	Representa una conexión con la base de datos	**createStatment()**	Crea un objeto Statement
Statement	Representa una instrucción SQL	**execute(String)**	Ejecuta la instrucción pasada como parámetro
ResultSet	Conjunto de registros devuelto por una instrucción SELECT	**first()**	Mueve el cursor al primer registro del conjunto
		getString(int)	Devuelve el valor de un campo del registro actual
		updateInt(int, int)	Asigna el valor entero indicado en el segundo parámetro al campo especificado
DatabaseMetadata	Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc	**getTables()**	Obtiene un ResultSet de las tablas que tenga una Base de Datos concreta.

JDBC: Conexión

- Dado que **JDBC** es independiente del gestor de base de datos utilizado, su diseño emplea principalmente interfaces, que especifican el conjunto de operaciones que aporta la librería. Esto tiene una serie de implicaciones prácticas:
  - El fabricante del **driver** debe proporcionar un conjunto de clases que implementen esas interfaces.
  - El código de la aplicación no tiene que conocer el driver que utiliza; escribes el código utilizando los interfaces aportados.
- Además **JDBC** incluye unas clases de uso general.
  - **DriverManager**. Para gestión de drivers **JDBC**. Sus métodos más importantes son **getConnection()** y **registerDriver()**.

JDBC: Conexión

- Pasos para conectar con una base de datos.

Paso	Objetivo	Procedimiento
1	Instalar el driver JDBC apropiado al gestor con el que trabajaremos	Conseguir la librería suministrada por el fabricante y hacerla accesible al *classpath* de la aplicación
2	Cargar el driver en memoria	Utilizar el método estático *forName()* de la clase *Class*
3	Obtener una conexión con la base de datos	Invocar a *DriverManager.getConnection()*, pasándole como parámetro la URL de la base de datos a la que deseamos conectarnos, el identificador del usuario, y la contraseña

JDBC: Conexión

- La instalación del **Driver**, no es mas que hacer accesible la librería **JDBC** del fabricante a nuestra aplicación, es decir añadirlo al **ClassPath**.
- La carga del **Driver** en memoria, se realiza para que el API de **JDBC**, conozca que **Driver** ha de emplear para comunicarse con la Base de Datos, esta carga puede realizarse de dos formas:
  - **Dinámica**: *Class.forName("<nombre de la clase Driver>");*
  - **Estática**: *System.setProperties("jdbc.drivers", "<nombre de la clase Driver>");*
- La clase **Driver**, ha de implementar el interfaz **java.sql.Driver**. La documentación del driver debe indicar cuál es esa clase.

JDBC: Conexión

- Una vez se ha cargado el Driver, solo queda obtener la conexión, para ello empleamos la clase **DriverManager**, y mas en concreto su método estático **getConnection()**.

```
java.sql.Connection conexion = java.sql.DriverManager.getConnection(url);
```

- El parámetro **url** que recibe el método **getConnection()** es un string que tendrá un formato concreto, que de forma genérica es el siguiente.

```
<protocolo>:<subprotocolo>:<subnombre>
```

JDBC: Conexión

- Es una cadena de texto formada por tres partes separadas con el carácter ':'. Cada parte será:
  - ***<protocolo>*** es siempre la cadena **jdbc**.
  - ***<subprotocolo>*** es un identificador del driver. La documentación del driver indicará qué subprotocolo le corresponde.
  - ***<subnombre>*** es la forma de identificar la base de datos concreta. Es una cadena cuya sintaxis depende de cada driver, y debe estar expresada en la documentación que éste facilita.
- Para cada proveedor, estos parámetros tomarán un valor distinto.

JDBC: Conexión

- Para el caso concreto del proveedor **Oracle**, este sería un ejemplo de conexión.

```
// (1/3): Instanciar el driver
Class.forName("oracle.jdbc.driver.OracleDriver");
// (2/3): Componer la URL
String url = "jdbc:oracle:thin:@192.168.111.128:1521:orcl"
// (3/3): Obtener la conexión
java.sql.Connection conexion= DriverManager.getConnection(url, "PRUEBAS", "PRUEBAS");
```

- En este caso se ve como para **Oracle**:
  - ***<protocolo>*** es **jdbc**.
  - ***<subprotocolo>*** es **oracle:thin**.
  - ***<subnombre>*** *es* ***@192.168.111.128:1521:orcl***

NOTA: Para MySQL sería **jdbc:mysql://192.168.111.128:3306/test**

JDBC: Crear Tablas

Paso	Objetivo	Procedimiento
Previo	Obtener una conexión con la base de datos	Obtener objeto java.sql.Connection.
1	Conseguir la sentencia SQL de creación de tabla	Elaborar, en base a la información sobre los datos que contendrá la tabla, la sentencia "CREATE TABLE..."
2	Conseguir un objeto JDBC para ejecutar sentencias DDL	Invocar al método *createStatement()* del objeto *Connection*, que representa la conexión con la base de datos; este método devuelve una referencia a un objeto *Statement*.
3	Ejecutar la sentencia	Invocar al método *executeUpdate()* del objeto *Statement*
4	Proceso posterior	Cerrar el *Statement* invocando a su método *close()*, para liberar memoria

JDBC: Crear Tablas

- A tener en cuenta los siguientes aspectos:
  - En general, **JDBC** se comunica con la base de datos utilizando **SQL**. A pesar de que es un estándar, distintos fabricantes pueden añadir extensiones, por lo que habrá que comprobar la sintaxis válida para cada caso.
  - La sentencia **SQL** no debe llevar ningún carácter de terminación (por ejemplo, punto y coma), como muchos gestores requieren en sus entornos.

JDBC: Crear Tablas

- Ejemplo de una tabla a crear.

Entrada	Cliente	DiaSemana	Tazas	Tipo
1	Alberto	LUN	1	Solo
2	Blanca	LUN	2	Capuchino
3	Carlos	MAR	2	Solo
4	Blanca	MAR	2	Capuchino

- La consulta **SQL** sería.

```
CREATE TABLE ConsumoDeCafe (
  Entrada INTEGER NOT NULL,
  Cliente VARCHAR2(50) NOT NULL,
  DiaSemana VARCHAR2(3) NOT NULL,
  Tazas INTEGER NOT NULL,
  Tipo VARCHAR2(15) NOT NULL,
  CONSTRAINT CONSUMODECAFE_PK PRIMARY KEY (Entrada)
  );
```

JDBC: Crear Tablas

- Código Java para crear la tabla anterior con **JDBC**.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS","PRUEBAS");
cn.setAutoCommit(false);
Statement st = cn.createStatement();
StringBuffer query = new StringBuffer("CREATE TABLE ConsumoDeCafe (");
query.append("Entrada INTEGER NOT NULL,");
query.append("Cliente VARCHAR2(50) NOT NULL,");
query.append("DiaSemana VARCHAR2(3) NOT NULL,");
query.append("Tazas INTEGER NOT NULL,");
query.append("Tipo VARCHAR2(15) NOT NULL,");
query.append("CONSTRAINT CONSUMODECAFE_PK PRIMARY KEY (Entrada)");
query.append(")");
st.executeUpdate(query.toString());
cn.commit();
st.close();
cn.close();
```

JDBC: Transacciones

- Al ejecutar una sentencia SQL en una base de datos, hay veces que la sentencia no debe tener efecto salvo que otras sentencias también se ejecuten con éxito, dado que de no ser así se podría producir una inconsistencia en los datos.
- Una transacción es un conjunto de una o más sentencias SQL que comprenden una unidad lógica de trabajo, de forma que cuando se ejecuta la transacción, se ejecutan en bloque todas las sentencias y, si cualquiera de ellas falla, ninguno de los cambios tiene efecto.

JDBC: Transacciones

- Algunos dialectos de SQL tienen sentencias específicas para iniciar y terminar transacciones, en general una transacción empieza al principio de un programa y continúa hasta que explícitamente el código indique que:
  - Se efectúe (**COMMIT**), lo que hace permanentes los cambios.
  - Se anule (**ROLLBACK**), con lo que se cancelarán todos los cambios.
- Tanto en uno como en otro caso, la transacción finaliza y en ese momento empieza una nueva transacción

JDBC: Transacciones

- Cuando creas una conexión en JDBC, tienes dos posibles modos de trabajar:
  - AutoCommit **activado**. Cada sentencia SQL se trata de forma individual como una transacción, que se comete de forma automática cuando se completa (cuando se ejecuta).
  - AutoCommit **desactivado**. Todas las sentencias SQL que ejecutas se incluyen en la transacción actual, y se cometerán juntas, como una unidad.

JDBC: Transacciones

- El objeto **java.sql.Connection**, tiene los siguientes métodos para el tratamiento de la transacción:
  - **setAutoCommit()**: Actia y desactiva AutoCommit.
  - **commit()**: Persiste los cambios.
  - **rollback()**: Cancela los cambios.

JDBC: Transacciones

```
…
cn.setAutoCommit(false);
StringBuffer query = new StringBuffer("UPDATE ConsumoDeCafe");
query.append(" SET Tazas = ? WHERE Entrada = ?");
PreparedStatement st = null;
try {
        st = cn.prepareStatement(query.toString());
        st.setInt(1, 2); // Asignar un 2 al parámetro 1 (Tazas)
        st.setInt(2, 1); // Asignar un 1 al parámetro 2 (Entrada)
        st.executeUpdate();
        st.setInt(1, 3); // Asignar un 2 al parámetro 1 (Tazas)
        st.setInt(2, 2); // Asignar un 1 al parámetro 2 (Entrada)
        st.executeUpdate();
        //lanzarException();
        cn.commit();
} catch (SQLException e) {
        e.printStackTrace();
        cn.rollback();
}
st.close();
cn.close();
```

JDBC: Tratamiento de errores

```
Connection cn = null;
try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
                "PRUEBAS","PRUEBAS");
        StringBuffer query = new StringBuffer("UPDATE ConsumoDeCafe");
        query.append(" SET Tazas = ? WHERE Entrada = ?");
        PreparedStatement st = null;
        st = cn.prepareStatement(query.toString());
        st.setInt(1, 21); // Asignar un 2 al parámetro 1 (Tazas)
        st.setInt(2, 1); // Asignar un 1 al parámetro 2 (Entrada)
        st.executeUpdate();
} catch (ClassNotFoundException cnf) {
        System.err.println("Error: no se pudo cargar el controlador de BD");
} catch (SQLException sq) {
        System.err.println("Error: en acceso a la BD");
} finally { // Método correcto de cerrar la conexión
        try {
                cn.close(); // Importante cerrar al menos este objeto
        } catch (SQLException ignorada) {}
}
```

JDBC: Insertar Registros

Paso	Objetivo	Procedimiento
Previo	Obtener una conexión con la base de datos	Obtener objeto java.sql.Connection
1	Construir la sentencia SQL para insertar un registro	Elaborar la sentencia “INSERT INTO...”
2	Conseguir un objeto JDBC para ejecutar la sentencia DML para insertar un registro	Invocar al método **createStatement()** del objeto Connection, que representa la conexión con la base de datos; este método devuelve una referencia a un objeto Statement.
3	Ejecutar la sentencia	Invocar al método **executeUpdate()** del objeto Statement
4	Proceso posterior	Cerrar el Statement invocando a su método **close()**, para liberar memoria.

JDBC: Insertar Registros

- Se puede observar que el procedimiento es similar al anterior, lo único que cambia realmente es la consulta **SQL** a lanzar, por lo que podemos concluir que el procedimiento es el mismo para cualquier sentencia que **no** devuelva datos:
  - Obtienes de la conexión un objeto **java.sql.Statement**.
  - Invocas a su método **executeUpdate()**.
- La consulta SQL para la inserción de un registro sería.

```
INSERT INTO ConsumoDeCafe
(Entrada, Cliente, DiaSemana, Tazas, Tipo)
VALUES (1, 'Alberto', 'LUN', 1, 'Solo');
```

JDBC: Insertar Registros

- Siguiendo con el ejemplo anterior, insertamos los registros correspondientes.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS","PRUEBAS");
cn.setAutoCommit(false);
Statement st = cn.createStatement();
// Insertar registro 1
StringBuffer query = new StringBuffer("INSERT INTO ConsumoDeCafe ");
query.append("(Entrada, Cliente, DiaSemana, Tazas, Tipo) ");
query.append("VALUES (1, 'Alberto', 'LUN', 1, 'Solo')");
st.executeUpdate(query.toString());
// Insertar registro 2
// Insertar registro 3
// Insertar registro 4
cn.commit();
st.close();
cn.close();
```

JDBC: Modificar Registros

Paso	Objetivo	Procedimiento
Previo	Obtener una conexión con la base de datos	Obtener objeto java.sql.Connection
1	Construir la sentencia SQL para modificar registros	Elaborar la sentencia “UPDATE...”
2	Preparar un objeto JDBC para ejecutar la sentencia DML para modificar registros	Invocar al método createStatement() del objeto Connection, que representa la conexión con la base de datos y devuelve una referencia a un objeto Statement.
3	Ejecutar la sentencia SQL de actualización	Invocar al método executeUpdate() del objeto Statement
4	Proceso posterior	Cerrar el Statement invocando a su método close(), para liberar recursos.

JDBC: Modificar Registros

- La consulta **SQL** para la modificación de un registro con “Tipo” igual a “Solo” cambiando el “Tipo” a “Base”, sería.

```
UPDATE ConsumoDeCafe
SET Tipo = 'Base'
WHERE Tipo = 'Solo‘;
```

JDBC: Modificar Registros

- Siguiendo con el ejemplo anterior, modificamos los registros con "Tipo" igual a "Solo", cambiando el "Tipo" a "Base".

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS", "PRUEBAS");
cn.setAutoCommit(false);
Statement st = cn.createStatement();
StringBuffer query = new StringBuffer("UPDATE ConsumoDeCafe");
query.append(" SET Tipo = 'Base'");
query.append(" WHERE Tipo = 'Solo'");
st.executeUpdate(query.toString());
cn.commit();
st.close();
cn.close();
```

JDBC: Eliminar Registros

Paso	Objetivo	Procedimiento
Previo	Obtener una conexión con la base de datos	Obtener objeto java.sql.Connection
1	Construir la sentencia SQL para eliminar registros	Elaborar la sentencia “DELETE FROM...”
2	Conseguir un objeto JDBC para ejecutar la sentencia DML para eliminar registros	Llamar al método createStatement() del objeto Connection. Este método devuelve una referencia a un objeto Statement.
3	Ejecutar la sentencia	Invocar al método executeUpdate() del objeto Statement
4	Proceso posterior	Cerrar el Statement invocando a su método close(), para liberar recursos y memoria.

JDBC: Eliminar Registros

- La consulta SQL para la eliminación de los registros que representen clientes que han tomado dos tazas de cualquier tipo de café en martes, sería.

```
DELETE FROM ConsumoDeCafe
WHERE DiaSemana = 'MAR'
AND Tazas = 2;
```

JDBC: Eliminar Registros

- Siguiendo con el ejemplo anterior, eliminamos los registros que representen clientes que han tomado dos tazas de cualquier tipo de café en martes.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS", "PRUEBAS");
cn.setAutoCommit(false);
Statement st = cn.createStatement();
StringBuffer query = new StringBuffer("DELETE FROM ConsumoDeCafe");
query.append(" WHERE DiaSemana = 'MAR'");
query.append(" AND Tazas = 2");
st.executeUpdate(query.toString());
cn.commit();
st.close();
cn.close();
```

JDBC: Consultar Tablas

Paso	Objetivo	Procedimiento
Previo	Obtener una conexión con la base de datos	Obtener objeto java.sql.Connection
1	Construir la sentencia SQL para consultar datos	Elaborar la sentencia “SELECT...”
2	Conseguir un objeto JDBC para ejecutar la sentencia	Invocar al método *createStatement()* del objeto *Connection*, este método devuelve una referencia a un objeto *Statement*
3	Ejecutar la sentencia SQL de consulta	Invocar al método *executeQuery()* del objeto Statement, que devuelve un ResultSet.
4	Avanzar al siguiente registro	Invocar al método *next()* del objeto ResultSet.
5	Obtener los datos del registro actual	Invocar a los métodos *getXXX()* del objeto ResultSet, dependiendo del tipo de dato
final	Liberar los recursos utilizados.	Cerrar el ResultSet, el Statement y la Connection invocando sus métodos *close()*

JDBC: Consultar Tablas

- Se puede observar que aunque el procedimiento es similar a los anteriores, además de cambiar la consulta SQL a lanzar, también se modifica el método invocado, siendo ahora **executeQuery()**, que además devuelve un objeto **java.sql.ResultSet**.
- La consulta SQL para la consulta de la tabla sería.

```
SELECT Cliente, Tazas, Tipo
FROM ConsumoDeCafe
WHERE DiaSemana = 'MAR‘;
```

JDBC: Consultar Tablas

- Siguiendo con el ejemplo anterior, consultamos los registros para el día de la semana “MAR”.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS","PRUEBAS");
Statement st = cn.createStatement();
StringBuffer query = new StringBuffer("SELECT Cliente, Tazas, Tipo");
query.append(" FROM ConsumoDeCafe");
query.append(" WHERE DiaSemana = 'MAR'");
ResultSet rs = st.executeQuery(query.toString());
while (rs.next()) {
                String cliente = rs.getString("Cliente"); // El argumento es el nombre de la columna
                int tazas = rs.getInt("Tazas");
                String tipo = rs.getString(3); // Ahora el argumento es su posición ordinal
                System.out.println("El cliente " + cliente + " " + "tomó el martes " + tazas + " tazas " +
"de café "    + tipo);
}
st.close();
cn.close();
```

JDBC: ResultSet

- Es un conjunto de registros de una o varias tablas, que el gestor de bases de datos devuelve como consecuencia de una consulta de información.
- Un conjunto de resultados no existe físicamente; es una "instantánea" en memoria de la situación de unos datos, que desaparece cuando terminamos de utilizarlo.
- En un **ResultSet**, uno de los registros es el activo; se puede consultar los valores de los campos del registro activo, que es referenciado por medio de un "puntero".

JDBC: ResultSet

- Un **ResultSet** es navegable; una vez obtenido, se puede recorrer moviendo el puntero al registro activo.
- Cuando **JDBC** devuelve un **ResultSet**, el puntero al registro activo está en la posición anterior al primer registro del resultado, por lo que lo primero que hay que hacer es avanzar el puntero, para que apunte al primer registro.
- En el anterior ejemplo, se ve en el paso 4 como antes de poder consultar los datos, se ejecuta el método ***next()*** sobre el ***ResultSet***, dado que inicialmente el puntero al registro activo está por delante del primer registro.

JDBC: ResultSet

- La llamada a **next()** devolverá ***true*** si ha avanzado a un registro válido, y ***false*** cuando se termine el resultado. Por eso se utiliza como condición en el bucle.
- En el bucle se extrae la información de cada registro.
- **ResultSet** tiene métodos para leer datos de cualquier tipo:
  - **getString()** para cadenas,
  - **getInt()** para enteros, etc.
- En esos métodos se puede dar como argumento
  - El nombre del campo
  - El índice dentro del conjunto de resultados, **empezando por 1**.

JDBC: PreparedStatement

- Internamente, al enviar una sentencia **SQL** a un gestor de bases de datos, éste debe realizar una serie de pasos para ejecutarla. En el primero de los pasos, el gestor hace un chequeo sintáctico de la sentencia y algunas otras tareas que normalmente se engloban con el nombre genérico de "***compilación***".
- Si se necesita ejecutar varias veces la misma sentencia SQL, (por ejemplo, en un bucle), o sentencias que difieren mínimamente, es interesante informar al gestor de esta situación, con lo que el proceso de compilación sólo se realiza una vez, y por tanto la eficiencia de su ejecución sería mayor.

JDBC: PreparedStatement

- Las ventajas de **PreparedStatement** frente a **Statement**, son:
  - La ejecución de un objeto **PreparedStatement** es mucho más rápida al tratarse de “sentencias pre-compiladas”
  - Pueden utilizarse parámetros en la sentencia, en forma de interrogaciones “?”, que las hace aún más flexibles
  - Ofrecen mayor seguridad frente a problemas como la temida “**inyección SQL**”.

JDBC: PreparedStatement

Paso	Objetivo	Procedimiento
previo	Obtener una conexión con la base de datos	Obtener objeto java.sql.Connection
1	Conseguir el objeto JDBC para ejecutar sentencias preparadas	Invocar al método *prepareStatement()* del objeto Connection, pasándole la sentencia SQL que se quiere ejecutar; este método devuelve una referencia a un objeto PreparedStatement
2	Proporcionar valores para los parámetros	Invocar a los métodos *set...()* del objeto PreparedStatement
3	Ejecutar la sentencia	Invocar al método *executeUpdate()* del objeto PreparedStatement
4	Proceso posterior	Cerrar el PreparedStatement invocando a su método *close()*, para liberar memoria

JDBC: PreparedStatement

- Modificar el numero de tazas que han tomado algunos de los clientes, por ejemplo al cliente con Id 1, ponerle 2 tazas y a los cliente con Id 2 y 3, ponerles 3 tazas.

```
cn.setAutoCommit(false);
StringBuffer query = new StringBuffer("UPDATE ConsumoDeCafe");
query.append(" SET Tazas = ? ");
query.append(" WHERE Entrada = ?");
PreparedStatement st = cn.prepareStatement(query.toString());
st.setInt(1, 2); // Asignar un 2 al parámetro 1 (Tazas)
st.setInt(2, 1); // Asignar un 1 al parámetro 2 (Entrada)
st.executeUpdate();
st.setInt(1, 3); // Asignar un 2 al parámetro 1 (Tazas)
st.setInt(2, 2); // Asignar un 1 al parámetro 2 (Entrada)
st.executeUpdate();
st.setInt(1, 3); // Asignar un 2 al parámetro 1 (Tazas)
st.setInt(2, 3); // Asignar un 1 al parámetro 2 (Entrada)
st.executeUpdate();
```

JDBC: Metadatos

- Los **metadatos** nos ofrecen información de como es la estructura de los objetos de la base de datos.
- Con **JDBC**, se pueden obtener los metadatos de la base de datos, o bien, de los resultados de una consulta.

JDBC: Metadatos de la Base de Datos

- Para obtener los metadatos de la base de datos, JDBC, nos provee la interfaz **java.sql.DatabaseMetaData**, esta interfaz tiene una serie de métodos, que nos proporcionan los metadatos de la base de datos a la que esta conectada, algunos de estos métodos son:
  - **getSchemas():** Ofrece información de los esquemas creados en la base de datos.
  - **getTypeInfo():** Ofrece información de los tipos de datos soportados por la base de datos.
  - **getTables():** Ofrece información de las tablas de la base de datos.

JDBC: Metadatos de la Base de Datos

- Un ejemplo de obtención de los metadatos de las tablas del esquema “PRUEBAS”.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS", "PRUEBAS");
DatabaseMetaData metaData = cn.getMetaData();
ResultSet rs = metaData.getTables(null, "PRUEBAS", null, new String[] { "TABLE" });
System.out.println("Tipos de datos de la base de datos");
while (rs.next()) {
                System.out.println(rs.getString("TABLE_SCHEM"));
                System.out.println(rs.getString("TABLE_NAME"));
                System.out.println(rs.getString("TABLE_TYPE"));
}
rs.close();
cn.close();
```

JDBC: Metadatos de un ResultSet

- Los metadatos de un **ResultSet** construido a partir de una consulta incluyen información sobre el número de columnas, sus tipos de datos y sus tamaños.
- Para obtener la información sobre un conjunto de resultados tendremos la interfaz **java.sql.ResultSetMetaData**, obtendremos este objeto a traves del método **getMetaData()** del objeto **ResultSet**.
- Este objeto nos provee de los siguientes metodos.
  - **getColumnName()**.
  - **getColumnTypeName()**.

JDBC: Metadatos de un ResultSet

- Un ejemplo de obtención de los metadatos “nombres” y “tipos de datos” de las columnas contenidas en el **ResultSet**.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn =
DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl", "PRUEBAS",
"PRUEBAS");
Statement st = cn.createStatement();
StringBuffer query = new StringBuffer("SELECT Cliente, Tazas, Tipo");
query.append(" FROM ConsumoDeCafe WHERE DiaSemana = 'MAR'");
ResultSet rs = st.executeQuery(query.toString());
ResultSetMetaData md = rs.getMetaData();
int nColumnas = md.getColumnCount();
System.out.println("Recibido ResultSet con " + nColumnas + " columnas");
for(int i = 1; i <= nColumnas; i++) {
            System.out.println("Columna " + i + ": " + md.getColumnName(i)
                  + " Tipo: " + md.getColumnTypeName(i));}
rs.close();
cn.close();
```

JDBC: Procedimientos almacenados

- Un procedimiento almacenado es un grupo de sentencias SQL que forman una unidad lógica y realizan una tarea en particular.
- Se puede emplear JDBC para crear o ejecutar procedimientos almacenados.
- La sintaxis para crear procedimientos almacenados es totalmente dependiente del gestor.

JDBC: Procedimientos almacenados

- La forma de ejecutar la sentencia de creación de un procedimiento, será igual a la de creación de una tabla.
- La secuencia de operaciones para ejecutar un procedimiento almacenado difiere dependiendo de si el procedimiento recibe o no parámetros, y si el resultado del procedimiento es un **ResultSet** o no.

JDBC: Procedimientos almacenados

- Creación de un procedimiento.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn =
DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS","PRUEBAS");
cn.setAutoCommit(false);
Statement st = cn.createStatement();
StringBuffer query = new StringBuffer("CREATE OR REPLACE PROCEDURE ");
query.append(" Actualiza_Tazas(in_entrada NUMBER, in_tazas NUMBER)");
query.append(" IS");
query.append(" BEGIN");
query.append(" UPDATE ConsumoDeCafe");
query.append(" SET Tazas = tazas");
query.append(" WHERE Entrada = entrada;");
query.append(" END Actualiza_Tazas;");
st.executeUpdate(query.toString());
cn.commit();
st.close();
cn.close();
```

JDBC: Procedimientos almacenados

- Invocar un procedimiento.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS","PRUEBAS");
cn.setAutoCommit(false);
StringBuffer query = new StringBuffer("{ call Actualiza_Tazas(?,?) }");
//Otra forma de invocarlo seria al estilo PL/SQL
//StringBuffer query = new StringBuffer("begin Actualiza_Tazas(?,?); end;");
CallableStatement st=cn.prepareCall(query.toString());
//Parametros de entrada
st.setInt(1, 1);//Entrada
st.setInt(2, 10);//Tazas
st.execute();
cn.commit();
st.close();
cn.close();
```

JDBC: Procedimientos almacenados

- Invocar un procedimiento con resultado de salida único.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS","PRUEBAS");
cn.setAutoCommit(false);
StringBuffer query = new StringBuffer("{ call Get_Tazas(?,?) }");
CallableStatement st=cn.prepareCall(query.toString());
//Parametro de salida
st.registerOutParameter(1, Types.INTEGER);//Tazas salida
//Parametro de entrada
st.setInt(2, 2);//Entrada
ResultSet rs = st.executeQuery();
System.out.println(st.getInt(1));
cn.commit();
rs.close();
st.close();
cn.close();
```

JDBC: Procedimientos almacenados

- Invocar un procedimiento con resultado de salida múltiple (**ResultSet**).

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@192.168.111.128:1521:orcl",
"PRUEBAS", "PRUEBAS");
StringBuffer query = new StringBuffer("{ call Get_Clientes(?,?) }");
CallableStatement st=cn.prepareCall(query.toString());
//Parametro de salida
st.registerOutParameter(1, OracleTypes.CURSOR);//Clientes salida
//Parametro de entrada
st.setInt(2, 3);//Tazas
st.executeQuery();
ResultSet rs = (ResultSet)st.getObject(1);
while (rs.next()){
        System.out.println(rs.getString("CLIENTE“) + "-“ + rs.getString("TIPO"));
}
rs.close();
st.close();
cn.close();
```

JDBC: Ampliación de ResultSet

- **ResultSet** nos permite además de la funcionalidad vista hasta ahora, que consistía en ir desplazando el cursor hacia adelante, las siguientes funcionalidades:
  - Ir hacia adelante o hacia atrás en un **ResultSet** o movernos a un fila específica.
  - Actualizar las tablas de la base datos utilizando métodos Java en lugar de utilizar comandos SQL.
  - Enviar múltiples secuencias SQL a la base de datos como una unidad, o **batch**.

JDBC: Navegar por un ResultSet

- Para navegar por un **ResultSet**, habrá que definir el **ResultSet** como navegable, esto se hace pasando unos parámetros al objeto **Statement**.
- Los parámetros que se le pueden pasar a **Statement** referentes a la navegación son:
  - **ResultSet.TYPE_SCROLL_INSENSITIVE**: No refleja los cambios que puede haber, mientras esté abierto, en las tablas a partir de las que se crea. Es como una "instantánea" de los datos que había en la tabla en el momento en que se abre.
  - **ResultSet.TYPE_SCROLL_SENSITIVE**: Si mientras está abierto se producen cambios (inserciones, modificaciones, borrados) en las tablas a partir de las que se creó, el **ResultSet** refleja esos cambios.

JDBC: Navegar por un ResultSet

- Métodos para desplazarse por un **ResultSet**:
  - **next().** Avanza al siguiente registro. Devuelve true si se posiciona sobre un registro válido, false si se posiciona más allá del último registro.
  - **previous().** Va al registro anterior al actual. Devuelve true si se posiciona sobre un registro válido, o false si se sale del **ResultSet** por arriba.
  - **first().** Va al primer registro del **ResultSet**.
  - **last().** Va al último registro del **ResultSet**.
  - **beforeFirst().** Pone el cursor inmediatamente antes del primer registro, al igual que cuando se inicia el **ResultSet**.
  - **afterLast().** Sitúa el cursor más allá del último registro.
  - **absolute().** Requiere un argumento entero. Si es positivo, va a la fila indicada; Si es negativo, se mueve ese número de filas desde el final, hacia el principio.
  - **relative().** También necesita un entero. Desplaza el cursor tantas filas como se indique, hacia adelante o atrás, si es negativo, desde la posición actual del cursor.
  - **isAfterLast()**. Indica si el cursor ha sobrepasado el último registro.
  - **isBeforeFirst()**. Indica si el cursor ha sobrepasado, por el principio, el primer registro; también es true cuando se ha abierto el **ResultSet** y todavía no se ha movido el cursor.

JDBC: Navegar por un ResultSet

- Ejemplos.

```
Statement stmt = oCn.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM ConsumoDeCafe");
```

```
// ... Previamente hemos obtenido la conexión en oCn
Connection oCn = DriverManager.getConnection(...)
// Obtener un objeto Statement para un ResultSet navegable:

Statement oStmt = oCn.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
// Abrir el ResultSet
ResultSet rs = oStmt.executeQuery("SELECT * FROM ConsumoDeCafe");
```

JDBC: Navegar por un ResultSet

- Ejemplos.

```
//recorrer todo el ResultSet del principio al fin
// Si está recién abierto, no hace falta la siguiente línea
rs.beforeFirst();
while (rs.next()) {
  // Tratar el registro activo
}
```

```
//recorrerlo desde el final
rs.afterLast();

while (rs.previous()) {
  // Tratar el registro activo
}
```

JDBC: Navegar por un ResultSet

- Ejemplos.

```
// Hace que el registro activo sea el primero
rs.first();
// Lleva el cursor al registro número 5
rs.absolute(5);
// Avanza el cursor dos registros hacia adelante
rs.relative(2);
// Mueve el cursor tres registros hacia atrás
rs.relative(-3);
// Pone el cursor en el antepenúltimo registro
rs.absolute(-2);
// Lleva el cursor al último registro
rs.last();
```

JDBC: Modificar datos en ResultSet

- Para modificar directamente los datos en el **ResultSet**, sin necesidad de ejecutar sentencias SQL, para ello hay que definir el **ResultSet** como modificable, esto se hace pasando unos parámetros al objeto **Statement**.
- Los parámetros que se le pueden pasar a **Statement** referentes a la modificación son:
  - **ResultSet.CONCUR_READ_ONLY:** Para crear un **ResultSet** de sólo lectura.
  - **ResultSet.CONCUR_UPDATABLE:** Para crear un **ResultSet** actualizable.

JDBC: Modificar datos en ResultSet

- Los pasos a seguir para modificar los datos de un **ResultSet** son:
  - Desplazarse hasta el registro a modificar.
  - Modificar el campo que se ha de modificar con los métodos **updateXXX()**, que tienen dos argumentos:
    - La columna a actualizar, que puede ser un índice (1,...), o el nombre de la columna.
    - El nuevo valor para la columna, cuyo tipo de datos (entero, cadena...) dependerá del método **updateXXX()**.
  - Persistir los cambios en la tabla, para lo cual existe el método **updateRow()**.

JDBC: Modificar datos en ResultSet

- Ejemplos.

```
// ... Previamente hemos obtenido la conexión en oCn
Connection oCn = DriverManager.getConnection(...)

// Obtener un objeto Statement para un ResultSet actualizable:
Statement oStmt = oCn.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);

// Abrir el ResultSet
ResultSet rs = oStmt.executeQuery("SELECT * FROM ConsumoDeCafe");
```

JDBC: Modificar datos en ResultSet

- Ejemplos.

```
// Obtener un objeto Statement para un ResultSet actualizable:
Statement oStmt = oCn.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);

// Abrir el ResultSet
ResultSet rs = oStmt.executeQuery("SELECT * FROM ConsumoDeCafe " +
                        "ORDER BY Entrada");
// Ir al último registro
rs.last();

// Establecer el valor de la columna "Tipo", a "Solo"
rs.updateString("Tipo", "Solo");

// Llevar los cambios a la tabla
rs.updateRow();

// Cerrar el ResultSet
rs.close();
// ...
```

JDBC: Añadir nuevos registros al ResultSet

- El procedimiento es igual al de la modificación, salvo en la elección del registro, dado que al ser nuevo, no se puede seleccionar uno existente, lo que se selecciona es la “fila de inserción”. Es una fila especial, una especie de buffer que se usa para poner los valores para las columnas de la nueva fila.

JDBC: Añadir nuevos registros al ResultSet

- Los pasos a seguir para la inserción de un nuevo registro son:
  - Desplazarse hasta la fila de inserción con el método **moveToInsertRow()**.
  - Insertar los nuevo valores en los campos con los métodos **updateXXX()**.
  - Persistir los cambios en la tabla, para lo cual existe el método **insertRow()**.
  - Se puede volver al registro anterior a la inserción con **moveToCurrentRow().**

JDBC: Añadir nuevos registros al ResultSet

- Ejemplos.

```
// Obtener un objeto Statement para un ResultSet actualizable:
Statement oStmt = oCn.createStatement(
                  ResultSet.TYPE_SCROLL_SENSITIVE,
                  ResultSet.CONCUR_UPDATABLE);

// Abrir el ResultSet
ResultSet rs = oStmt.executeQuery("SELECT * FROM ConsumoDeCafe");

// Insertar el nuevo registro
rs.moveToInsertRow();
rs.updateString("Cliente", "José Luis");
rs.updateString("DiaSemana", "JUE");
rs.updateInt("Tazas", 2);
rs.updateString("Tipo", "Capuchino");
rs.insertRow();

// Cerrar el ResultSet
rs.close();
// ...
```

JDBC: Eliminar registros del ResultSet

- El procedimiento es igual al de la modificación.
- Los pasos a seguir para el borrado de un registro son:
  - Desplazarse hasta el registro a eliminar.
  - Eliminar el registro invocando el método **deleteRow()**.

JDBC: Eliminar registros del ResultSet

- Ejemplos.

```
// Obtener un objeto Statement para un ResultSet actualizable:
Statement oStmt = oCn.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);

// Abrir el ResultSet
ResultSet rs = oStmt.executeQuery("SELECT * FROM ConsumoDeCafe " +
                "ORDER BY Entrada");

// Eliminar el tercer registro
rs.absolute(3);
rs.deleteRow();

// Cerrar el ResultSet
rs.close();
// ...
```

JDBC: características de la extensión estándar de jdbc

El paquete **javax.sql** es una extensión estándar al lenguaje Java. La especificación final está incluida en la versión **3.0 de JDBC**. Aunque no va a ser tratada en este curso, merece la pena conocer algunas características de la extensión del JDBC 2.0.

- **Rowsets** Este objeto encapsula un conjunto de filas de una hoja de resultados y podría mantener abierta la conexión con la base de datos o desconectarse de la fuente. Un **rowset** es un componente Java Bean; puede ser creado en el momento del diseño y puede utilizarse con otros Java Beans en una herramienta visual.
- **JNDI** para Nombrar Bases de Datos. El interface JNDI (Nombrado y Direccionado) de Java hace posible conectar a una base de datos utilizando un nombre lógico en lugar de codificar un nombre de base de datos y de driver.
- **Connection Pooling** Un Connection Pool es un caché para conexiones frecuentes que pueden ser utilizadas y reutilizadas, esto recorta la sobrecarga de crear y destruir conexiones a bases de datos.
- **Soporte de Transacción Distribuida**. Este soporte permite al driver JDBC soportar el protocolo estándar de dos-fases utilizados en el API Java Transaction (JTA). Esta característica facilita el uso de las funcionalidades

JDBC: características de la extensión estándar de jdbc

- **DataSource** El acceso a bases de datos puede realizarse a través de un servidor de aplicaciones. Este servidor no es necesario para utilizar JDBC, sin embargo nos ofrece unas ventajas interesantes.

Por una parte nos abstrae de tener que abrir nosotros las conexiones, los Statements, etc., es decir, nos evita tener que incluir código de programación. Además se pueden configurar los parámetros de acceso para el contenedor maneje todo el proceso de conexión accediendo a un fichero simple xml, como web.xml.

```
<resource-ref>
      <description>Conexion Base Datos</description>
      <res-ref-name>jdbc/artDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
 </resource-ref>
```

JDBC: características de la extensión estándar de jdbc

El código java de un programa java (un Servlet en el ejemplo), que quisiera acceder a la base de datos por medio de un DataSource sería así:

```
public class PoolDBServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;
	private DataSource fuenteDatos;
	private Context context;

	public void init(ServletConfig config) {
		System.out.println(" > El servlet PoolDBServlet se ha iniciado");
		try {
			context = new InitialContext();
			System.out.print(" > Cargando un contexto JNDI...");
			fuenteDatos = (DataSource) context.lookup("java:comp/env/jdbc/artDB");
			System.out.println(" hecho!");
		} catch (NamingException e) {
			System.out.println(" > Falló el contexto JNDI...");
			System.exit(1);   // Si no funciona el contexto, abortamos el programa y salimos
			e.printStackTrace();
		} // … resto del código fuente
```

Fin