Tema 10: JDBC

Hablar a una Base de Datos

Hay tres conceptos importantes cuando se trabaja con una base de datos:

- Crear una conexión a la base de datos.
- Crear una declaración para ejecutar en la base de datos.
- Recuperar un conjunto de datos que representa los resultados.

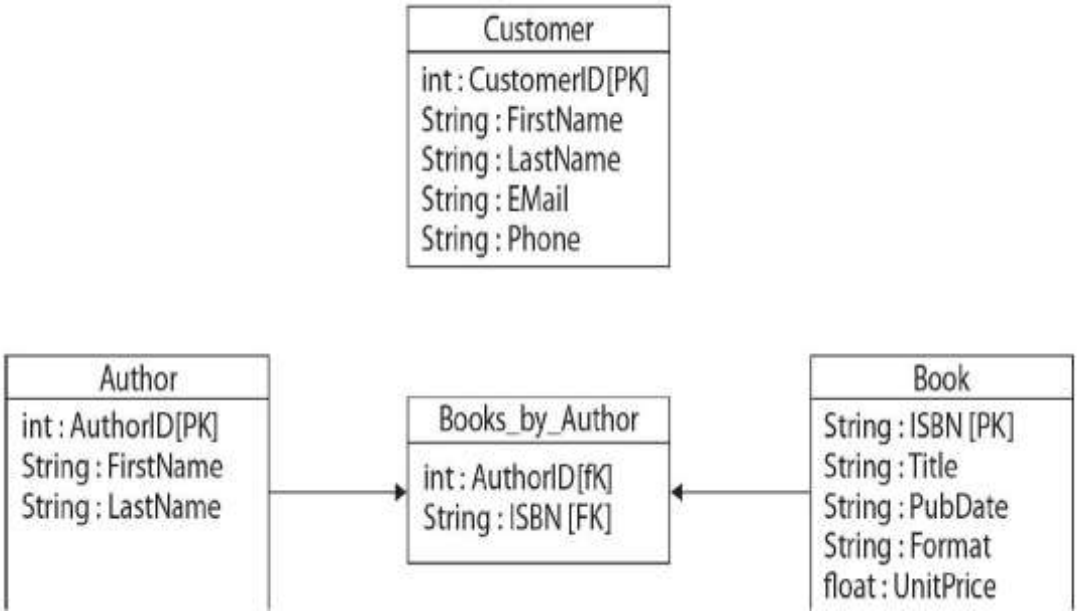
Antes de que podamos comunicarnos con el software que administra la base de datos, antes de que podamos enviarle una consulta, necesitamos establecer una conexión con el RDBMS en sí mismo. Hay muchos tipos diferentes de conexiones, y muchas tecnologías subyacentes para describir la conexión en sí, pero en general, para comunicarse con un RDBMS, necesitamos abrir una conexión usando una IP dirección y número de puerto a la base de datos. Una vez que hayamos establecido el conexión, necesitamos enviarle algunos parámetros (como un nombre de usuario y contraseña) para autenticarnos como un usuario válido del RDBMS. Finalmente, asumiendo que todo salió bien, podemos enviar consultas a través de la conexión. Una vez que hemos establecido una conexión, podemos usar algún tipo de aplicación (generalmente proporcionada por el proveedor de la base de datos) para enviar sentencias de consulta a la base de datos, ejecutarlas en la base de datos y obtener un conjunto de resultados devueltos.

Consultas SQL

"CRUD"	SQL Command	Example SQL Query	Expressed in English
Create	INSERT	`INSERT INTO Expenses VALUES ('April', 231.21, 29.87, 97.00, 45.08)`	Add a new row (April) to expenses with the following values....
Read (or Find)	SELECT	`SELECT * FROM Expenses WHERE Month="February"`	Get me all of the columns in the Expenses table for February.
Read All	SELECT	`SELECT * FROM Expenses`	Get me all of the columns in the Expenses table.
Update	UPDATE	`UPDATE Expenses SET Phone=32.36, EatingOut=111.08 WHERE Month='February'`	Change my Phone expense and EatingOut expense for February to....
Delete	DELETE	`DELETE FROM Expenses WHERE Month='April'`	Remove the row of expenses for April.

Los comandos SQL, como SELECT, INSERT, UPDATE, etc., son insensibles a las mayúsculas y minúsculas. Todas las bases de datos conservan las mayúsculas y minúsculas cuando una cadena es delimitado, es decir, cuando está entre comillas. Entonces, una cláusula SQL que usa las comillas simples o dobles para delimitar un identificador preservarán las mayúsculas y minúsculas del identificador.

Nuestra base de datos de ejemplo



Este es el contenido de la **tabla Customer**:

CustomerID	FirstName	LastName	Email	Phone
5000	John	Smith	john.smith@verizon.net	555-340-1230
5001	Mary	Johnson	mary.johnson@comcast.net	555-123-4567
5002	Bob	Collins	bob.collins@yahoo.com	555-012-3456
5003	Rebecca	Mayer	rebecca.mayer@gmail.com	555-205-8212
5006	Anthony	Clark	anthony.clark@gmail.com	555-256-1901
5007	Judy	Sousa	judy.sousa@verizon.net	555-751-1207
5008	Christopher	Patriquin	patriquinc@yahoo.com	555-316-1803
5009	Deborah	Smith	debsmith@comcast.net	555-256-3421
5010	Jennifer	McGinn	jmcginn@comcast.net	555-250-0918

Este es el contenido de la **tabla Books**:

ISBN	Title	PubDate	Format	Price
142311339X	The Lost Hero (Heroes of Olympus, Book 1)	2010-10-12	Hardcover	10.95
0689852223	The House of the Scorpion	2002-01-01	Hardcover	16.95
0525423656	Crossed (Matched Trilogy, Book 2)	2011-11-01	Hardcover	12.95
1423153627	The Kane Chronicles Survival Guide	2012-03-01	Hardcover	13.95
0439371112	Howliday Inn	2001-11-01	Paperback	14.95
0439861306	The Lightning Thief	2006-03-12	Paperback	11.95
031673737X	How to Train Your Dragon	2010-02-01	Hardcover	10.95
0545078059	The White Giraffe	2008-05-01	Paperback	6.95
0803733428	The Last Leopard	2009-03-05	Hardcover	13.95
9780545236	Freaky Monday	2010-01-15	Paperback	12.95

Este es el contenido de la **tabla Authors**:

AuthorID	FirstName	LastName
1000	Rick	Riordan
1001	Nancy	Farmer
1002	Ally	Condie
1003	Cressida	Cowell
1004	Lauren	St. John
1005	Eoin	Colfer
1006	Esther	Freisner
1007	Chris	D'lacey
1008	Mary	Rodgers
1009	Heather	Hatch

Este es el contenido de la **tabla Books_by_Author**:

AuthorID	ISBN
1000	142311339X
1001	0689852223
1002	0525423656
1000	1423153627
1003	031673737X
1004	0545078059
1004	0803733428
1008	9780545236
1009	9780545236

Interfaces principales de la API JDBC

El propósito de una base de datos relacional es realmente triple:

- Proporcionar almacenamiento para datos en tablas.
- Proporcionar una manera de crear relaciones entre los datos.
- Proporcionar un idioma que se pueda utilizar para obtener, actualizar, eliminar y crear datos.

El propósito de **JDBC (Java Database Connectivity)** es proporcionar una interfaz de programación de aplicaciones (API) para que los desarrolladores de Java escriban aplicaciones Java que puedan acceder y manipular bases de datos relacionales y usar **SQL** para realizar operaciones **CRUD**. **JDBC** también puede ser utilizado para acceder a sistemas de archivos y fuentes de datos orientadas a objetos.
La clave es que la API proporciona una vista abstracta de una conexión de base de datos, declaraciones y conjuntos de resultados. Estos conceptos se representan en la API como interfaces en el **paquete java.sql**: **Connection, Statement** y **ResultSet,** respectivamente:

- El interfaz **java.sql.Connection** define el contrato para un objeto que representa la conexión con un sistema de base de datos relacional.

- El interfaz **Statement** proporciona una abstracción de la funcionalidad necesitada para obtener una instrucción SQL para ejecutar en una base de datos.

- El interfaz **ResultSet** es una funcionalidad de abstracción necesaria para procesar un conjunto de resultados (la tabla de datos) que se devuelve de la consulta SQL cuando la consulta involucra una sentencia SQL **SELECT**.

La colección de las clases de implementación se llama **JDBC driver**. Un **driver JDBC** ("d" minúscula) es la colección de clases necesarias para soportar a la API, mientras que Driver ("D" mayúscula) es uno de los implementaciones requeridas en un driver.
El driver JDBC normalmente es proporcionado por un vendedor en un archivo JAR o ZIP. Las clases de implementación del controlador deben cumplir con un conjunto mínimo de requisitos para cumplir con JDBC:

- Implementar completamente las interfaces: **java.sql.Driver, java.sql.DatabaseMetaData, java.sql.ResultSetMetaData.**
- Implementar la interfaz **java.sql.Connection**.
- Implemenar **java.sql.Statement** y **java.sql.PreparedStatement.**
- Implementar las **interfaces java.sql.CallableStatement** si la base de datos admite procedimientos almacenados.
- Implementar la interfaz **java.sql.ResultSet.**

Conectarse a una base de datos usando DriveManager

No todos los tipos definidos en la **API JDBC** son interfaces. Una clase importante para JDBC es la clase **java.sql.DriverManager**. Esta clase concreta es se utiliza para interactuar con un controlador JDBC y devolver instancias de **Connection**. Conceptualmente, la forma en que esto funciona es mediante el uso de un **patrón de diseño Factory**. En un patrón Factory, se utiliza una clase concreta con métodos estáticos para crear instancias de objetos que implementan una interfaz.

```
public interface Vehicle {
  public void start( );
  public void stop( );
}
```

Necesitamos una implementación de Car para usar este contrato. Entonces diseñamos un Car:

```
package  com.us.automobile;
public class Car implements Vehicle {
   public void start( ) {  }
   public void stop( ) {  }
}

public class MyClass {
  public static void main(String[ ] args) {
     Vehicle Ferrari = new com.us.automobile.Car( );
     ferrari.start( );
  }
}
```

Sin embargo, aquí sería mejor una factory, de esa manera, no necesitamos saber nada sobre la implementación actual, y, como veremos con **DriverManager**, podemos usar métodos de la factory para determinar dinámicamente qué implementación usar en tiempo de ejecución.

```
public class MyClass {
  public static void main(String[ ] args) {
    Vehicle Ferrari = CarFactory.getVehicle(“Ferrari”);
     ferrari.start( );
  }
}
```

La factory en este caso podría crear un Car diferente basado en la cadena pasada al método estático getVehicle( ), algo como esto:

```
public class CarFactory {
  public static Vehicle getVehicle(String type){
    //…crea una instancia de un objeto que representa el tipo de Car pasado como
    //argumento
  }
}
```

DriverManager utiliza este patrón de factory para "construir" una instancia de un objeto **Connection** pasando una cadena a su método **getConnection( ).**

La clase DriverManager

La clase **DriverManager** es una clase concreta de utilidad en la API JDBC con métodos estáticos. La clase **DriverManager** se llama así porque administra qué implementación del controlador JDBC obtenemos cuando solicitamos una instancia de una conexión a través del método getConnection( ).Hay varios métodos getConnection sobrecargados, pero todos comparten un parámetro común: una URL de cadena. Un patrón para getConnection es:

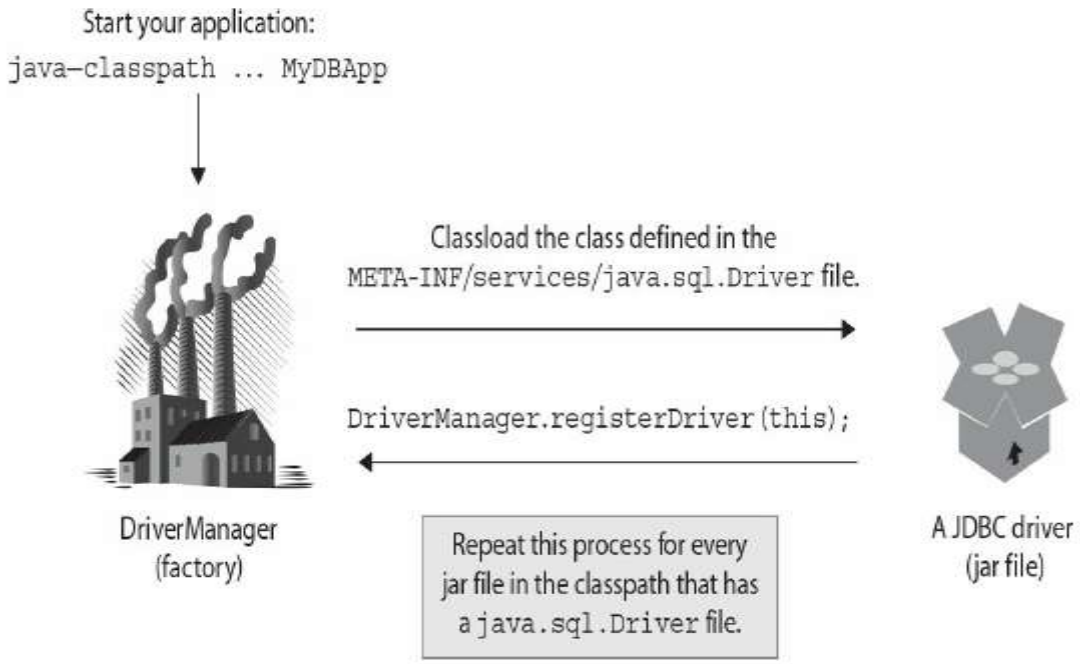
```
DriverManager.getConnection(String url, String username, String password);
```

Por ejemplo:

```
String url = “jdbc:derby://localhost:1521/BookSellerDB”;
String user = “bookguy”;
String pwd = “$3lleR”;
try{
   Connection conn = DriverManager.getConnection(url, user, pwd);
} catch(SQLException se){ }
```

Si la URL es incorrecta, o el nombre de usuario y / o contraseña son incorrectos, entonces el método lanzará una SQLException.

Como se registran los Drivers JDBC con el DriverManager



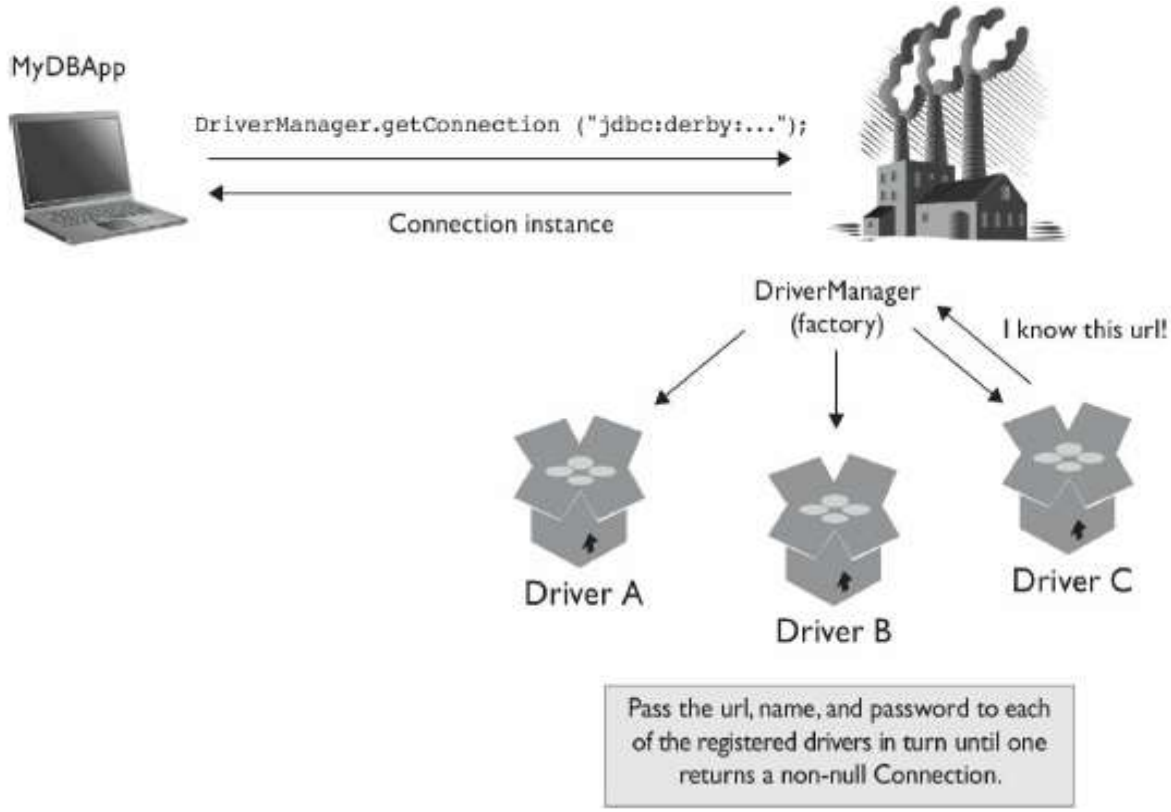
Primero, uno o más controladores JDBC, en un archivo **JAR** o **ZIP**, se incluyen en el classpath de nuestra aplicación. La clase **DriverManager** usa un mecanismo proveedor de servicio para buscar en el classpath cualquier archivo JAR o ZIP que contenga un archivo llamado **java.sql.Driver** en la carpeta **META-INF/services** del jar o del zip del driver. Este es simplemente un archivo de texto que contiene el nombre completo de la clase que el proveedor usó para implementar la interfaz **jdbc.sql.Driver**.
El DriverManager intentará cargar la clase que encontró en el archivo java.sql.Driver usando el cargador de clases:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

Cuando se carga la clase del driver, se ejecuta su bloque de inicialización estático. Por la especificación JDBC, una de las primeras actividades de una instancia de controlador es "autorregistrarse" con la clase DriverManager invocando un método estático en DriverManager. El código (menos manejo de errores) sería algo así:

```
public class ClientDriver implements java.sql.Driver {
  static {
      ClientDriver driver = new ClientDriver( );
      DriverManager.registerDriver(driver);
  }
  //....
}
```

Esto registra (almacena) una instancia de la clase **Driver** en el **DriverManager.** Ahora, cuando nuestra aplicación invoca el DriverManager.getConnection( ) y pasa una URL JDBC, nombre de usuario y contraseña del método, **DriverManager** simplemente invoca al método connect( ) en el driver registrado. Si la conexión fue exitosa, el método devuelve una instancia de objeto **Connection** a **DriverManager**, que, a su vez, nos lo devuelve. Si hay más de un controlador registrado, **DriverManager** llama a cada de los controladores a su vez e intenta obtener un objeto **Connection** de ellos:



Resumiendo:

- La **JVM** carga la clase **DriverManager**, una clase concreta en el **API JDBC**.
- La clase **DriverManager** carga cualquier instancia de clases que encuentre en el **META-INF/services/java.sql.Driver** archivo de archivos **JAR / ZIP** en el classpath.
- Las clases Driver llaman a **DriverManager.register(this)** para registrarse con el **DriverManager**.
- Cuando el método **DriverManager.getConnection (String url)** es invocado, **DriverManager** invoca el método connect( ) de cada una de esas instancias de **Driver** registradas con la cadena **URL**.
- El primer **Driver** que crea con éxito una conexión con la URL devuelve una instancia de un objeto **Connection** a la invocación del método **DriverManager.getConnection.**

La URL JDBC

La URL JDBC es lo que se usa para determinar qué implementación de driver utilizar para una **Connection** dada.

```
jdbc:derby://localhost:1521/BookSellerDB
```

La primera parte, jdbc, simplemente identifica que esta es una URL JDBC (versus HTTP u otra cosa). La segunda parte indica que el proveedor del driver es derby La tercera parte indica que la base de datos está en el host local de esta máquina (dirección IP 127.0.0.1), en el puerto 1521, y la parte final indica que estamos interesados en la base de datos BookSellerDB. Los vendedores pueden modificar la URL para definir características para una implementación de controlador particular. El formato de la URL JDBC es:

```
jdbc:<subprotocolo>:<subnombre>
```

En general, el subprotocolo es el nombre del proveedor, por ejemplo, **jdbc:derby, jdbc:mysql, jdbc:oracle,…**
El campo de subnombre es donde las cosas se vuelven un poco más específicas del proveedor. Algunos los proveedores usan el subnombre para identificar el nombre de host y el puerto, seguidos de un nombre de la base de datos. Por ejemplo:

```
jdbc:derby://localhost:1521/MyDB
jdbc:mysql://localhost:3306/MyDB
```

Otros proveedores pueden usar el subnombre para identificar información adicional de contexto sobre el driver. Por ejemplo:

```
jdbc:oracle:thin:@//localhost:1527/MyDB
```

Versiones de implementación del Driver JDBC

- Antes de que podamos comenzar a trabajar con **JDBC,** crear consultas y obtener resultados, primero debe establecer una conexión.

- Para establecer una conexión, debe tener un driver o controlador **JDBC**.

- Si nuestro driver **JDBC** es un driver **JDBC 3.0,** entonces deberemos cargar explícitamente el driver en nuestro código usando Class.forName( ) y la ruta totalmente calificada de la clase de implementación de el Driver.

- Si nuestro driver **JDBC** es un driver **JDBC 4.0**, simplemente incluiremos el driver (jar o zip) en el classpath.

Enviar consultas y leer resultados de la base de datos

Para consultar la base de datos y devolver todos los Clientes en la base de datos, escribiríamos algo como el ejemplo que se muestra a continuación:

```
import static java.lang.System.*;
String url = “jdbc:derby://localhost:1521/BookSellerDB”;
String user = “bookguy”;
String pwd = “$3lleR”;
try{
Connection conn = DriverManager.getConnection(url, user,pwd);
Statement stmt = conn.createStatement( );
String query = “SELECT * FROM Customer”;
ResultSet rs = stmt.executeQuery(query);
while( rs.next( )) {
  out.print(rs.getInt(“CustomerID” + “  “));
  out.print(rs.getString(“FIrstName” + “  “));
  out.print(rs.getString(“LastName” + “  “));
  out.print(rs.getString(“Email” + “  “)); ());
  out.print(rs.getString(“Phone”));
 }
} catch( SQLException se){ }
```

Lo que se hace aquí es lo siguiente:

- **Obtener conexión** Estamos creando una instancia de objeto **Connection** usando la información que necesitamos para acceder a la Base de datos de libros de Bob (almacenada en un Base de datos relacional de Java, BookSellerDB, y se accede a través de credenciales "bookguy" con una contraseña "$3lleR").

- **Crear una declaración (statement)** Estamos utilizando la conexión para crear un objeto **Statement**. El objeto **Statement** maneja pasar cadenas a la base de datos como consultas para que la base de datos se ejecute.

- **Ejecutar consulta** Estamos ejecutando la cadena de consulta en la base de datos y devolviendo un objeto **ResultSet**.

- **Resultados del proceso** Estamos iterando a través de las filas del conjunto de resultados, cada llamada a next( ) nos mueve a la siguiente fila de resultados.

- **Imprimir columnas** Estamos obteniendo los valores de las columnas en el fila de conjunto de resultados actual e imprimiéndolos por la salida estándar.

- **Catch SQLException** Todas las invocaciones de métodos de la API JDBC lanzan SQLException. Se puede lanzar una excepción SQLException cuando un método

se usa incorrectamente o si la base de datos no responde. Por ejemplo, se produce una excepción SQLException si la URL JDBC, el nombre de usuario o la contraseña no es válida O intentamos consultar una tabla que no existe. O la base de datos ya no es accesible porque la red se cayó o la base de datos se desconectó.

Statements

Utilizando la instancia de **Connection** que recibimos de **DriverManager**, podemos obtener una instancia de un objeto que implementa el interfaz **Statement.** Hay varias formas de métodos **Statement:** aquellos que devuelven un conjunto de resultados y aquellos que devuelven un entero que refleja el estado.

- **Construyendo y usando Statements**

Como no todas las sentencias SQL devuelven resultados, el objeto **Statement** proporciona varios métodos diferentes para ejecutar comandos SQL. Algunos comandos SQL no devuelven un conjunto de resultados, sino que devuelven un entero. Por ejemplo, **SQL INSERT, UPDATE** y **DELETE**, o cualquiera de las instrucciones del lenguaje de definición (**DDL**) como **CREATE TABLE**, devuelven el número de filas afectadas por la consulta o 0. Veamos cada uno de los métodos de ejecución en detalle.

```
public ResulSet executeQuery(String sql) throws SQLException
```

Este método se usa cuando sabemos que queremos devolver resultados; estamos consultando la base de datos para una o más filas de datos. Por ejemplo:

```
ResulSet rs = stmt.executeQuery("SELECT * from Customer");
```

Hay que tener en cuenta que la declaración del método incluye **throws SQLException**. Esto significa que este método debe llamarse en un bloque try-catch o debe llamarse en un método que también lance **SQLException**.

```
public int executeUpdate(String sql) throws SQLException
```

Este método se usa para una operación SQL que afecta a una o más filas y no devuelve resultados, por ejemplo, consultas SQL INSERT, UPDATE, DELETE y consultas DDL. Estas declaraciones no devuelven resultados, pero sí un recuento del número de filas afectadas por la consulta SQL. Para consultas DDL, el valor de retorno es 0.

```
String q = "UPDATE Book SET UnitPrice = 8.95 WHERE UnitPrice < 8.95 AND
Format ='HardCover'";
int numRows = stmt.executeUpdate(q);
```

```
public boolean execute(String sql) throws SQLException
```

Este método se usa cuando no estamos seguros de cuál será el resultado, tal vez la consulta devuelva un conjunto de resultados y tal vez no. Este método puede usarse para ejecutar un consulta cuyo tipo puede no ser conocido hasta el tiempo de ejecución, por ejemplo, uno construido en código. El valor de retorno es **true** si la consulta resultó en un resultado establecido y **false** si la consulta dio como resultado un recuento de actualizaciones o ningún resultado. Sin embargo, con mayor frecuencia, este método se usa al invocar un procedimiento almacenado.

```
ResulSet rs;
int numRows;
boolean status = stmt.execute(“”);
if(status) {
  rs= stmy.getResultSet( );
} else {
  numRows = stmt.getUpdateCount( );
  if(numRows == -1) {
    out.println(“No results”);
  } else {
    out.println(numRows + “ rows affected.”);
  }
}
```

El método **getResultSet( )** se utiliza para recuperar resultados cuando el el método **execute( )** devuelve true, y el método **getUpdateCount( )** se usa para recuperar el recuento cuando el método **execute( )** devuelve false.

```
public ResultSet getResultSet( ) throws SQLException
```

Si el valor booleano del método execute( ) devuelve true, entonces hay un conjunto de resultados. Para obtener el conjunto de resultados llamaremos al método getResultSet( ) en el objeto de **Statement**. Entonces podremos procesar el objeto **ResultSet**.

```
public int getUpdateCount( ) throws SQLException
```

Si el valor booleano devuelto por el método execute( ) es false, hay un recuento de filas, y el método devolverá el número de filas afectadas. Un valor de retorno de –1 indica que no hay resultados.

```
int numRows = stmt.getUpdateCount( );
if (numRows == -1) {
  out.println(“No results”);
} else {
  out.println(numRows + “ rows affected.”);
}
```

Method (Each Throws SQLException)	Description
`ResultSet executeQuery(String sql)`	Execute a SQL query and return a `ResultSet` object, i.e., SELECT commands.
`int executeUpdate(String sql)`	Execute a SQL query that will only modify a number of rows, i.e. INSERT, DELETE, or UPDATE commands.
`boolean execute(String sql)`	Execute a SQL query that may return a result set OR modify a number of rows (or do neither). The method will return true if there is a result set or false if there may be a row count of affected rows.
`ResultSet getResultSet()`	If the return value from the `execute()` method was true, you can use this method to retrieve the result set from the query.
`int getUpdateCount()`	If the return value from the `execute()` method was false, you can use this method to get the number of rows affected by the SQL command.

ResultSets

El objeto **ResultSet** representa los resultados de la consulta: todos los datos en cada fila basados en columnas. Usando los métodos definidos en la interfaz **ResultSet**, podemos leer y manipular los datos.

Una cosa importante a tener en cuenta es que un **ResultSet** es una copia de los datos de la base de datos de la instancia en el momento en el que la consulta se ejecutó. A menos que seamos la única persona que utiliza la base de datos, debemos suponer siempre que la tabla o tablas de base de datos subyacentes sobre las que se ejecutó la consulta que origino el **ResultSet** podrían haber sido cambiadas por algún otro usuario o aplicación.

- **Moviéndonos hacia adelante en un ResultSet**

El objeto **ResultSet** mantiene un cursor o un puntero, a la fila actual de los resultados. Cuando el objeto ResultSet es devuelto por la consulta, el cursor aún no apunta a una fila de resultados: el cursor apunta hacia arriba de la primera fila. Para obtener los resultados de la tabla, siempre debemos llamar al método **next( )** en el objeto **ResultSet** para mover el cursor hacia adelante a la primera fila de datos. Por defecto, un objeto ResultSet es solo lectura (los datos en las filas no se pueden actualizar) y solo puede mover el cursor hacia adelante.

```
public boolean next( )
```

El método **next( )** mueve el cursor una fila hacia adelante y devuelve **true** si el cursor ahora apunta a una fila de datos en el **ResultSet.** Si el cursor apunta más allá de la última fila de datos como resultado de la método **next( )** (o si **ResultSet** no contiene filas), el valor de retorno es **false**.

```
String query = “SELECT FirstName. LastName, Email FROM Customer WHERE
LastName LIKE ‘C%’”;
ResultSet rs = stmt.excuteQuery(query);
while(rs.next( )) {     //….    }
```

- **Leyendo datos desde un ResultSet**

Para llevar los datos de cada columna a nuestra aplicación de Java, debemos usar un método **ResultSet** para recuperar cada uno de los SQL valores de columna en un tipo Java apropiado. Entonces SQL INTEGER, para ejemplo, puede leerse como una primitiva Java int, SQL VARCHAR puede leerse como un String Java, SQL DATE puede leerse como un objeto java.sql.Date, y así sucesivamente. **ResultSet** define varios otros tipos también, pero si la base de datos o el driver admite todos los tipos definidos por la especificación depende del proveedor de la base de datos.

SQL Type	Java Type	ResultSet get methods
BOOLEAN	boolean	getBoolean(String columnName) getBoolean(int columnIndex)
INTEGER	int	getInt(String columnName) getInt(int columnIndex)
DOUBLE, FLOAT	double	getDouble(String columnName) getDouble(int columnIndex)
REAL	float	getFloat(String columnName) getFloat(int columnIndex)
BIGINT	long	getLong(String columnName) getLong(int columnIndex)
CHAR, VARCHAR, LONGVARCHAR	String	getString(String columnName) getString(int columnIndex)
DATE	java.sql.Date	getDate(String columnName) getDate(int columnIndex)
TIME	java.sql.Time	getTime(String columnName) getTime(int columnIndex)
TIMESTAMP	java.sql.Timestamp	getTimestamp(String columnName) getTimestamp(int columnIndex)
Any of the above	java.lang.Object	getObject(String columnName) getObject(int columnIndex)

```
String query = “SELECT Title, PubDate, Price FROM Book”;
ResultSet rs = stmt.executeQuery(query);
while (rs.next( )) {
  String title = rs.getString(“Title”);
  Date PubDate = rs.getDAte(“PubDate”);
  float price = rs.getFloat(“Price”);
   //…
}
```

Hay que tener en cuenta que aunque aquí se recuperaron los nombres de columna de la fila ResultSet en el orden en que se solicitaron en la consulta SQL, podrían haber sido procesados en cualquier orden. ResultSet también proporciona un método sobrecargado que toma un índice entero valor para cada uno de los tipos de SQL. Este valor es la posición entera del columna en el conjunto de resultados, numerada del 1 al número de columnas devueltas.

```
String title = rs.getString(1);
Date PubDate = rs.getDAte(2);
float price = rs.getFloat(3);
```

Es importante recordar que los índices de las columnas empiezan en 1.

Los métodos de ResultSet más utilizados son:

- **public boolean getBoolean(String columnLabel)**

Los valores booleanos rara vez se devuelven en consultas SQL, y algunas bases de datos pueden no admite un tipo BOOLEAN de SQL, así que deberemos consultar a nuestro proveedor de base de datos.

```
if (rs.getBoolean("CURR_EMPLOYEE")) {
   // …
}
```

- **public double getDouble(String columnLabel)**

Este método se recomienda para devolver el valor almacenado en la base de datos como tipos DOUBLE y FLOAT de SQL.

```
double cartTotal = rs.getDouble("CartTotal");
```

- **public int getInt(String columnLabel)**

Este método se recomienda para devolver valores almacenados en la base de datos como tipos INTEGER de SQL.

```
int authorID = rs.getInt("AuthorID");
```

- **public float getFloat(String columnLabel)**

Este método recupera el valor de la columna como un **float** de Java. Se recomienda para los tipos REAL de SQL.

```
float Price = rs.getFloat("UnitPrice");
```

- **public long getLong(String columnLabel)**

Este método recupera el valor de la columna como un long de Java. Se recomienda para los tipos BIGINT de SQL.

```
long socialSecurityNumber = rs.get("SocSecNum");
```

- **public java.sql.Date getDate(String columnLabel)**

Este método recupera el valor de la columna como un objeto **Date** de Java. Hay que tener en cuenta que **java.sql.Date** extiende a **java.util.Date**. Una diferencia entre los dos es que el método **toString( ) de java.sql.Date** devuelve una cadena de fecha en la forma:"yyyy mm dd". Este método se recomienda para los tipos de DATE de SQL.

```
java.sql.Date pubDate = rs.getDate("PubDAte");
```

- **public String getString(String columnLabel)**

Este método recupera el valor de la columna como un objeto **String** de Java. Es bueno para lectura de columnas SQL con tipos CHAR, VARCHAR y LONGVARCHAR.

```
String lastName = rs.getString("LastName");
```

- **public java.sql.Time getTime(String columnLabel)**

Este método recupera el valor de la columna como un objeto Java **Time**. Como java.sql.Date, esta clase extiende java.util.Date, y su método toString( ) devuelve una cadena de tiempo en la forma: "hh: mm: ss". Este método está diseñado para leer el tipo TIME de SQL.

```
java.sql.Time time = rs.getTime("FinishTime");
```

- **public java.sql.Timestamp getTimestamp(String columnLabel)**

Este método recupera el valor de la columna como un objeto Timestamp. Su método toString( ) formatea el resultado en la forma: **yyyy-mm-dd, hh: mm: ss.fffffffff**, donde **fffffffff** es nanosegundos. Este método es recomendado para leer tipos TIMESTAMP de SQL.

```
java.sql.Timestamp timestamp = rs.getTimestamp("ClockInTime");
```

- **public java.lang.Object getObject(String columnLabel)**

Este método recupera el valor de la columna como un objeto Java. Se puede usar como método de propósito general para leer datos en una columna.

```
Object o = rs.getObject("AuthorID");
if (o instanceof java.lang.Integer) {
   int id = ((Integer)o).intValue( );   }
```

Obteniendo información sobre un ResultSet

Cuando escribimos una consulta con una cadena, como hemos visto en los ejemplos hasta ahora, sabemos el nombre y el tipo de las columnas devueltas. Sin embargo, ¿qué sucede cuando deseamos permitir que los usuarios construyan dinámicamente la consulta? Es posible que no siempre sepamos de antemano cuántas columnas son devueltas y el tipo y nombre de las columnas devueltas.

Afortunadamente, la clase **ResultSetMetaData** fue diseñada para proporcionar justo esa información. Usando **ResultSetMetaData**, podemos obtener importante información sobre los resultados devueltos por la consulta, incluido el número de columnas, el nombre de la tabla, el nombre de la columna y el nombre de la clase de columna(la clase Java que se usa para representar esta columna cuando la columna es devuelto como un objeto).

```
String query = "SELECT AuthorID Author";
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData( );
rs.next( );
int colCount = rsmd.getColumnCount( );
out.println("Column Count: " + colCount);
for (int =1; I <= colCount; i++) {
   out.println("Table Name: " + rsmd.getTableName(i));
   out.println("Column Name: " + rsmd.getColumnName(i));
   out.println("Column Size: " + rsmd.getColumnDisplaySize(i));  }
```

ResultSetMetaData se usa a menudo para generar informes, así que aquí están los métodos de uso más común.

- **public int getColumnCount( ) throws SQLException**

Devuelve el recuento del número de columnas devueltas por la consulta.

```
try {
  con = DriverManager.getConnection(…);
  stmt = con.createStatement( );
  String query = "SELECT * FROM Author";
  ResultSet rs = stmt.executeQuery(query);
  ResultSetMetaData rsmd = rs.getMetaData( );
  int columnCount = rsmd.getColumnCount( );
  ….
} catch(SQLException se) { }
```

- **public String getColumnName(int column) throws SQLException**

Este método devuelve un String con el nombre de esta columna.

```
String colData;
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData( );
int cols = rsmd.getColumnCount( );
for(int i = 1; i<= cols; i++) {
  out.print(rsmd.getColumnName(i)+ " ");
}
out.println( );
while(rs.next( )) {
  for(int i = 1; i<= cols; i++) {
      if( rs.getObject(i) != null) {
        colData = rs.getObject(i).toString( );
      } else {
        colData = "NULL";
      }
     out.print(colData);
  }
  out.println( );
}
```

- **public String getTableName(int column) throws SQLException**

El método devuelve un String con el nombre de la tabla a la que pertenece esta columna.

```
ResultSetMetaData rsmd = rs.getMetaData( );
int cols = rsmd.getColumnCount( );
for(int i = 1; i<= cols; i++) {
  out.print(rsmd.getTableName(i)+ ":" + rsmd.getColumnName(i) + " ");
}
```

- **public int getColumnDisplaySize(int column) throws SQLException**

Este método devuelve un entero del tamaño de la columna. Esta información es útil para determinar el número máximo de caracteres que puede contener una columna (si es un tipo VARCHAR) y el espacio que se requiere entre columnas para un informe.

Imprimiendo un informe

Usando los métodos que hemos discutido hasta ahora, aquí hay un código que produce un bonito informe de una consulta:

```
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData( );
int cols = rsmd.getColumnCount( );
String col, colData;
for(int i = 1; i<= cols; i++) {
  col = leftJustify( rsmd.getColumnName(i),  rsmd.getColumnDisplaySize(i));
  out.print(col);
}
out.println( );
 while(rs.next( )) {
  for(int i = 1; i<= cols; i++) {
     if(rs.getObject(i) != null) {
        colData = rs.getObject(i).toString( );
      } else {
        colData = “NULL”;
      }
      col = leftJustify(colData, rsmd.getColumnDisplaySize(i)));
      out.print(col);
   }
 out.println( );
}
```

El método leftJustify:

```
public static String leftJustify(String s, int n) {
  if (s.length( ) <= n) n++;
  return String.format(“%1$-” + n + ”s”, s);
}
```

Este método usa el método String format( ) y el indicador "-" (guión) para devolver una cadena que es justificada a la izquierda con espacios. La parte %1$ indica que la bandera debe aplicarse al primer argumento. Lo que estamos construyendo es una cadena de formato de forma dinámica. Si el tamaño de visualización de la columna es 20, la cadena de formato será% 1 $ -20s, que dice:
"Imprime el argumento pasado (el primer argumento) a la izquierda con un ancho de 20 y usar una conversión de cadena".

Por otra parte, las bases de datos pueden almacenar valores NULL. Si el valor de una columna es NULL, el objeto devuelto en el método rs.getObject () es un nulo de Java. Entonces tenemos que probar nulo para evitar obtener una excepción de puntero nulo cuando se ejecute el método toString( ).

Moviéndonos por un ResultSet

La especificación JDBC define tipos int estáticos adicionales (mostrados a continuación) que permiten a un desarrollador mover el cursor hacia adelante, hacia atrás y hacia una posición específica en el conjunto de resultados. Además, el conjunto de resultados puede modificarse mientras está abierto y los cambios pueden ser escritos en la base de datos.

Para crear un conjunto de resultados que utiliza cursores posicionables y/o admite actualizaciones, debemos crear un **Statement** con el tipo de desplazamiento apropiado y la configuración de concurrencia, y luego usar esa instrucción para crear el objeto **ResultSet.** Hay tres tipos de cursor **ResultSet:**

- **TYPE_FORWARD_ONLY** El valor predeterminado para un **ResultSet**: el cursor se mueve hacia adelante solo a través de un conjunto de resultados.

- **TYPE_SCROLL_INSENSITIVE** Se puede mover la posición del cursor en el resultado hacia adelante o hacia atrás, o posicionar un cursor en una ubicación particular. Cualquier cambio realizado en los datos subyacentes (la base de datos en sí misma) no se reflejan en el conjunto de resultados. En otras palabras, el conjunto de resultados no tiene que "mantener el estado" con la base de datos. Este tipo es generalmente soportado por bases de datos.

- **TYPE_SCROLL_SENSITIVE** Se puede mover el cursor hacia adelante o hacia atrás en los resultados, o posicionarlo en una ubicación particular. Cualquier cambio realizado en los datos subyacentes se refleja en el conjunto resultante. Como podemos imaginar, esto es difícil de implementar y, por lo tanto, no se implementa en una base de datos o en el driver JDBC con mucha frecuencia.

JDBC proporciona dos opciones para la concurrencia de datos con un conjunto de resultados:

- **CONCUR_READ_ONLY** Este es el valor predeterminado para la concurrencia del conjunto de resultados. Cualquier conjunto de resultados abierto es de solo lectura y no se puede modificar o cambiado.

- **CONCUR_UPDATABLE** Un conjunto de resultados se puede modificar a través de los métodos ResultSet mientras el conjunto de resultados está abierto.

Debido a que una base de datos y un controlador JDBC no son necesarios para admitir el movimiento del cursor y las actualizaciones concurrentes, el JDBC proporciona métodos para consultar la base de datos y el driver utilizando el objeto **DatabaseMetaData** para determinar si nuestro driver admite estas capacidades.

```
Connection con = DriverManager.getConnection(…);
DataBaseMetaData dbnd = con.getMetaData( );

if (dbmd.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY)) {
   out.print(“Supports TYPE_FORWARD_ONLY”);
      if (dbmd.supportsResultSetConcurrency(
         ResultSet.TYPE_FORWARD_ONLY,
         ResultSet.CONCUR_UPDATABLE)) {
         out.println(“and supports CONCUR_UPDATABLE”);
      }
}

if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)) {
   out.print(“Supports TYPE_SCROLL_INSENSITIVE”);
      if (dbmd.supportsResultSetConcurrency(
         ResultSet.TYPE_SCROLL_INSENSITIVE,
         ResultSet.CONCUR_UPDATABLE)) {
         out.println(“and supports CONCUR_UPDATABLE”);
      }
}

if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE)) {
   out.print(“Supports TYPE_SCROLL_SENSITIVE”);
      if (dbmd.supportsResultSetConcurrency(
         ResultSet.TYPE_SCROLL_SENSITIVE,
         ResultSet.CONCUR_UPDATABLE)) {
         out.println(“and supports CONCUR_UPDATABLE”);
      }
}
```

Podemos determinar con qué tipo de cursor y concurrencia se crea el **Statement**, pero una vez creado, no puede cambiar el tipo de cursor o concurrencia de un objeto **Statement** existente.

Puede darse el caso de que el driver determine que la base de datos no soporta uno o ambos de los argumentos de configuración que elegimos y que lance una advertencia y (en silencio) vuelva a su configuración predeterminada, si no están soportados.

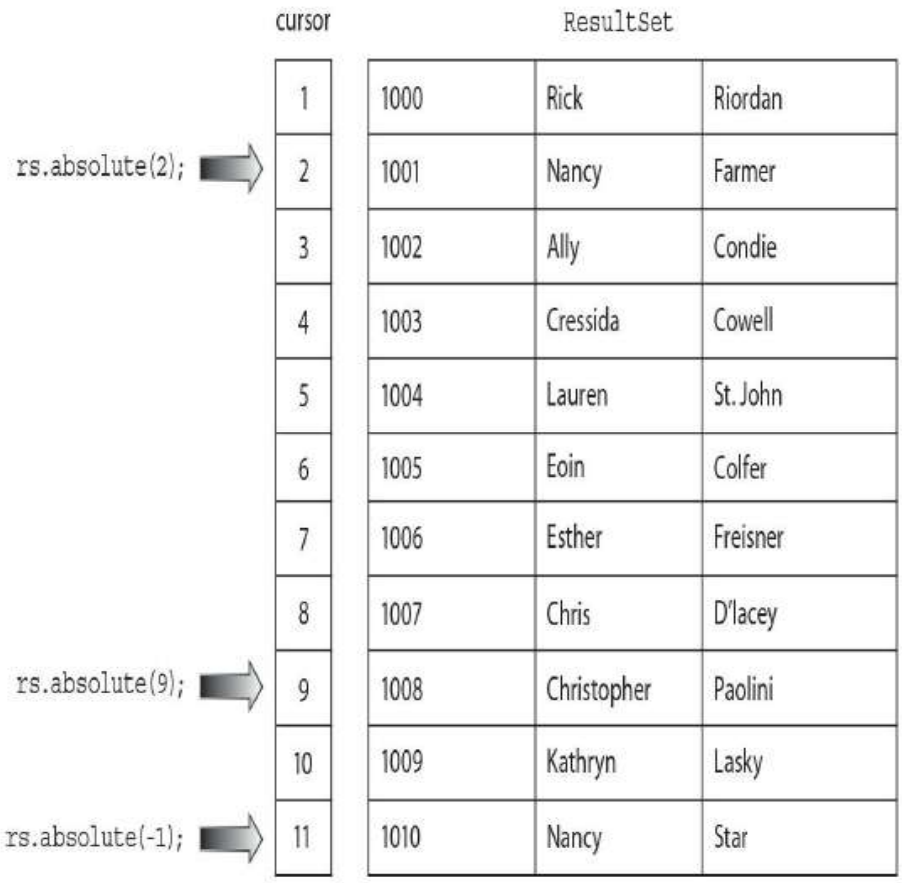
A continuación, una lista con los métodos que utilizamos para cambiar la posición del cursor en un **ResultSet.**

Method	Effect on the Cursor and Return Value
`boolean next()`	Moves the cursor to the next row in the `ResultSet`. Returns `false` if the cursor is positioned beyond the last row.
`boolean previous()`	Moves the cursor backward one row. Returns `false` if the cursor is positioned before the first row.
`boolean absolute(int row)`	Moves the cursor to an absolute position in the `ResultSet`. Rows are numbered from 1. Moving to row 0 moves the cursor to before the first row. Moving to negative row numbers starts from the last row and works backward. Returns `false` if the cursor is positioned beyond the last row or before the first row.
`boolean relative(int row)`	Moves the cursor to a position relative to the current position. Invoking `relative(1)` moves forward one row; invoking `relative(-1)` moves backward one row. Returns `false` if the cursor is positioned beyond the last row or before the first row.
`boolean first()`	Moves the cursor to the first row in the `ResultSet`. Returns `false` if there are no rows in the `ResultSet` (empty result set).
`boolean last()`	Moves the cursor to the last row in the `ResultSet`. Returns `false` if there are no rows in the `ResultSet` (empty result set).
`void beforeFirst()`	Moves the cursor to before the first row in the `ResultSet`.
`void afterLast()`	Moves the cursor to after the last row in the `ResultSet`.

- **public boolean absolute(int row) throws SQLException**

Este método coloca el cursor en un número de fila absoluto. Pasar 0 como argumento de fila coloca el cursor antes de la primera fila. Al pasar un valor negativo, como -1, coloca el cursor en la posición después de la última fila menos uno, en otras palabras, la última fila. Si intentamos posicionar el cursor más allá de la última fila, digamos en la posición 22 en un resultado de 19 filas, el cursor se colocará más allá de la última fila. El método **absolute( )** devuelve verdadero si el cursor fue exitoso posicionado dentro del **ResultSet** y falso si el cursor terminó antes de la primera o después de la última fila.

```
ResultSet rs = stmt.executeQuery(query);
for(int i=1; I += 2)) {
  if(rs.absolute(i)){
      // El método absolute mueve a la fila pasada como valor
      // entero y devuelve true si el movimiento tuvo éxito.
   } else {
       break;
   }
}
```



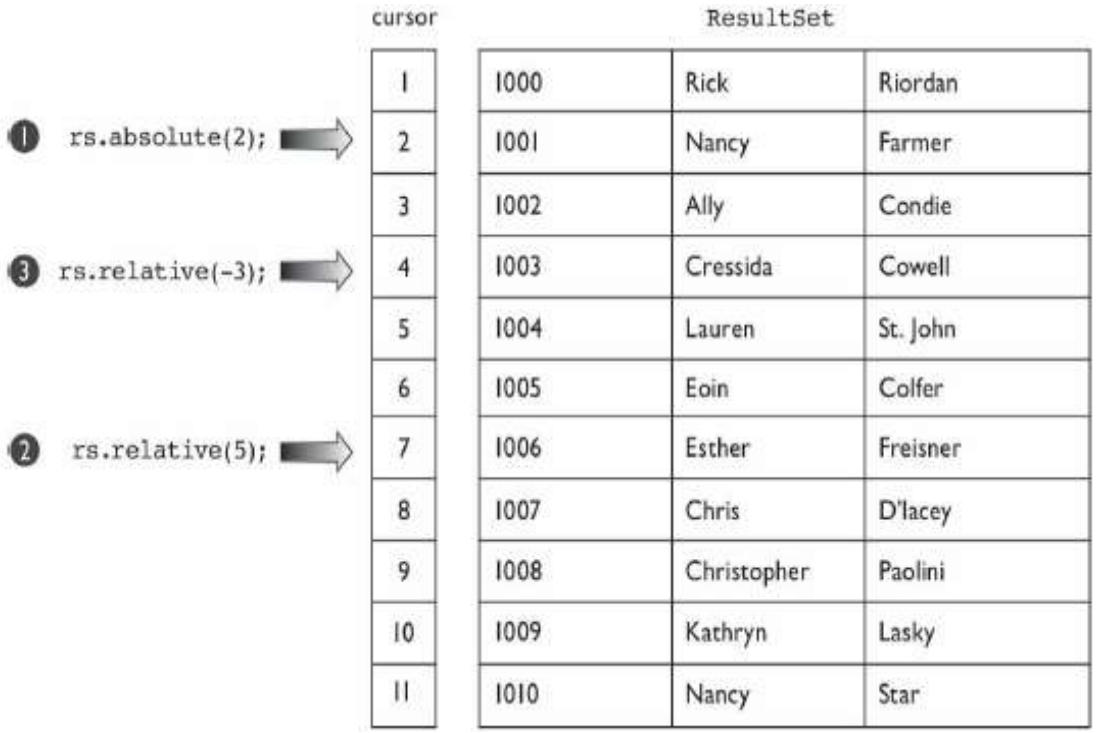
- **public int getRow( ) throws SQLException**

Este método devuelve la posición actual de la fila como un entero positivo (1 para la primera fila, 2 para la segunda, y así sucesivamente) o 0 si no hay una fila actual (el cursor está antes de la primera fila o después de la última fila). Este es el único método de este conjunto de métodos de cursor que es opcionalmente soportado por los **ResultSets** TYPE_FORWARD_ONLY.

- **public boolean relative(int rows) throws SQLException**

relative( ) posicionará el cursor antes o después de la posición actual según el número de filas pasadas al método. Entonces, si el cursor está en la fila 15 de un ResultSet de 30 filas, llamando a relative(2) colocará el cursor en la fila 17, y luego llamando al relative(-5) posiciona el cursor en la fila 12.

Al igual que el posicionamiento absoluto, intentar colocar el cursor más allá de la última fila o antes de la primera fila simplemente da como resultado que el cursor esté después de la última fila o antes de la primera fila, respectivamente, y el método devuelve falso.



- **public boolean previous( ) throws SQLException**

El método **previous( )** funciona exactamente igual que el método **next( ),** solo que va hacia atrás una posición a través del **ResultSet**.

- **public void afterLast( ) throws SQLException**

Este método posiciona el cursor después de la última fila. Usando este método y luego el método previous( ), se puede iterar a través de un ResultSet a la inversa. Por ejemplo:

```
public void showFlippedResultSet(ResultSet rs) throws SQLException {
  rs.afterLast( );
  while(rs.prvious( )) {
     // …
  }
}
```

Al igual que **next( ),** cuando **previous( )** realiza una copia de seguridad antes de la primera fila, el método devuelve **false**.

- **public void beforeFirst( ) throws SQLException**

Este método devolverá el cursor a la posición que tenía cuando se creó **ResultSet** por primera vez y fue devuelto por un objeto **Statement**.

```
rs.beforeFirst( );  // Posiciona el cursor antes de la primera fila
```

- **public boolean first( ) throws SQLException**

El método **first( )** coloca el cursor en la primera fila. Es el equivalente de llamar absolute(1). Este método devuelve true si el cursor se movió a una fila válida y false si el **ResultSet** no tiene filas.

```
if( !rs.first( )) {
  out.println("No rows in the result set.");
}
```

- **public boolean last( ) throws SQLException**

El método **last( )** posiciona el cursor en la última fila. Este método es equivalente a llamar a absolute(-1). Este método devuelve true si el cursor se movió a un valor de fila válido y false si el **ResultSet** no tiene filas.

```
if( !rs.last( )) {
  out.println("No rows in the result set.");
}
```

Un par de notas sobre las excepciones lanzadas por todos estos métodos:

- Estos métodos generarán una excepción **SQLException** si el tipo de ResultSet es **TYPE_FORWARD_ONLY**, si ResultSet es cerrado , o si ocurre un error en la base de datos.

- Estos métodos generarán una **SQLFeatureNotSupportedException** si el driver JDBC no soporta al método. Esta excepción es una subclase de **SQLException**.

- La mayoría de estos métodos no tienen efecto si ResultSet no tiene filas (por ejemplo, un ResultSet devuelto por una consulta que no devolvió filas).

Los siguientes métodos devuelven un valor booleano para permitirnos "probar" la posición actual del cursor sin mover el cursor:

- **isBeforeFirst( )** Verdadero si el cursor se coloca antes de la primera fila.

- **isAfterLast( )** Verdadero si el cursor se posiciona después de la última fila.

- **isFirst( )** Verdadero si el cursor está en la primera fila.

- **isLast( )** Verdadero si el cursor está en la última fila.

```
public static int getRowCount (ResultSet rs) throws SQLException {
  int rowCount = -1;
  int currRow =  0;
  if (rs != null) {
     currRow = rs.getRow( );
    if (rs.isAfterLast( )) { currRow = -1; }
    if(rs.last( )) {
      rowCount = rs.getRow( );
        if (currRow == -1) { rs.afterLast( ); }
        else if (currRow == 0) { rs.beforeFirst( ); }
        else {  rs.absolute(currRow); }
     }
   }
   return rowCount;
  }
```

Actualizando ResultSets

Además de solo devolver los resultados de una consulta, un objeto **ResultSet** puede usarse para modificar el contenido de una tabla en base de datos, incluida la actualización de filas existentes, eliminar filas existentes y agregar nuevas filas.

```
// Tenemos una conexión y estamos en un bloque try-catch…
Statement stmt= conn.createStatement( );
String query = “UPDATE Book SET UnitPrice = 11.95 AND Format = ‘Hardcover’ ”;
int rowsUpdated = stmt.executeUpdate(query);
```

¿Qué pasaría sí solo deseásemos aumentar el precio de los bestsellers, en lugar de el de todos los libros? Cuando crea un Statement con concurrencia establecida en **CONCUR_UPDATABLE,** podemos modificar los datos en un conjunto de resultados y luego aplicar los cambios a la base de datos sin tener que emitir otra consulta.

Tendríamos que obtener los valores de la base de datos utilizando un SELECT, luego almacenar los valores en una matriz indexada de alguna manera (tal vez con la clave primaria), luego construir las cadenas de comando UPDATE apropiadas y llamar a executeUpdate( ) una vez para cada fila. Otra opción es actualizar el ResultSet directamente.
Además de los métodos getXXXX que analizamos para ResultSet (métodos que obtienen valores de columna como enteros, objetos de fecha, Strings, etc…), hay un método updateXXXX equivalente para cada tipo. Y, al igual que los métodos getXXXX, los métodos updateXXXX pueden tomar un nombre de columna como un String o como un índice de columna entera.

```
// Tenemos una conexión y estamos en un bloque try-catch…
Statement stmt= conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
String query = "SELECT UnitPrice from Book  WHERE Format = 'Hardcover'";
ResultSet rs = stmt.executeQuery(query);
while(rs.next( )) {
  if (rs.getFloat("UnitPrice" == 10.95f);
    rs.updateFloat("UnitPrice", 11.95f);
    rs.updateRow( );
  }
 }
```

Observemos que después de modificar el valor de UnitPrice usando el método updateFloat( ), llamamos al método updateRow( ). Este método escribe la fila actual en la base de datos. Este enfoque de dos pasos asegura que todos los cambios se realizan en la fila antes de que la fila se escriba en la base de datos. Y podemos cambiar de opinión con una llamada al método cancelRowUpdates( ).

Métodos utilizados con ResultSet actualizables:

Method	Purpose
`void updateRow()`	Updates the database with the contents of the current row of this `ResultSet`.
`void deleteRow()`	Deletes the current row from the `ResultSet` and the underlying database.
`void cancelRowUpdates()`	Cancels any updates made to the current row of this `ResultSet` object. This method will effectively undo any changes made to the `ResultSet` row. If the `updateRow()` method was called before `cancelRowUpdates`, this method will have no effect.
`void moveToInsertRow()`	Moves the cursor to a special row in the `ResultSet` set aside for performing an insert. You need to move to the insert row before updating the columns of the row with update methods and calling `insertRow()`.
`void insertRow()`	Inserts the contents of the insert row into the database. Note that this method does not change the current `ResultSet`, so the `ResultSet` should be read again if you want the `ResultSet` to be consistent with the contents of the database.
`void moveToCurrentRow()`	Moves the cursor back to the current row from the insert row. If the cursor was not on the insert row, this method has no effect.

- **public void updateRow( ) throws SQLException**

Este método actualiza la base de datos con el contenido de la fila actual del ResultSet. Hay un par de advertencias para este método. Primero, el ResultSet debe ser de una sentencia SQL SELECT de una sola tabla (una sentencia SQL que incluye un JOIN o una instrucción SQL con dos tablas no se puede actualizar). En segundo lugar, el método updateRow( ) debe ser llamado antes de pasar a la siguiente fila. De lo contrario, las actualizaciones de la fila actual pueden perderse.

El uso típico de este método es actualizar el contenido de una fila usando los métodos updateXXXX( ) apropiados y luego actualizar la base de datos con el contenido de la fila utilizando el método updateRow( ).

```
rs.updateFloat("UnitPrice", 11.95f);   // Establece el precio a 11.95
rs.updateRow( );                       // Actualiza la fila en la BDD.
```

- **public boolean rowUpdate( ) throws SQLException**

Este método devuelve true si la fila actual se actualizó. Hay que tener en cuenta que no todas las bases de datos pueden detectar actualizaciones Sin embargo, JDBC proporciona un método en DatabaseMetaData para determinar si las actualizaciones son detectables, **DatabaseMetaData.updatesAreDetected(int type),** donde el tipo es uno de los tipos ResultSet (TYPE_SCROLL_INSENSITIVE, por ejemplo).

```
if(rs.rowUpdated( )) { out.println("Row: " + rs.getRow( ) + " updated."); }
```

- **public void cancelRowUpdates( ) throws SQLException**

Este método le permite "deshacer" los cambios realizados en la fila. Este método es importante porque los métodos updateXXXX no deberían ser llamados dos veces en la misma columna. En otras palabras, si establecemos el valor de UnitPrice en 11.95 en el ejemplo anterior y luego decidimos cambiar el precio nuevamente a 10.95, llamar al método updateFloat( ) nuevamente puede conducir a resultados impredecibles. Entonces el mejor enfoque es llamar a cancelRowUpdates( ) antes de cambiar el valor de un columna por segunda vez.

```
boolean priceRollback = … ;
while (rs.next( )) {
  if (rs.getFloat("UnitPrice") == 10.95f) {
     rs.updateFloat("UnitPrice", 11.95f);
  }
  if (priceRollback) {
    rs.cancelRowUpdates( );
  } else {
    rs.updateRow( );
  }
}
```

- **public void deleteRow( ) throws SQLException**

Este método eliminará la fila actual del ResultSet y de la base de datos subyacente. La fila de la base de datos es eliminada (similar al resultado de una instrucción DELETE).

```
rs.last( );
rs.deleteRow( );  // Elimina la última fila
```

Lo que sucede con ResultSet después de un método deleteRow( ) depende de si ResultSet puede detectar eliminaciones. Y esta habilidad depende del driver JDBC. El interfaz DatabaseMetaData se puede usar para determinar si el ResultSet puede detectar eliminaciones:

```
int type = ResultSet.TYPE_SCROLL_INSENSITIVE;
DatabaseMetaData dbmd = conn.getMetaData( );
if (dbmd.deletesAreDetected(type)) {
  while(rs.next( )) {
    if (rs.rowDeleted( )) {
      continue;
    } else {
       // process the row
    }
} else {
  // Cerramos el ResultSet
}
```

Eliminar la fila actual no mueve el cursor, permanece en la fila actual: por lo tanto, si eliminamos la fila 1, el cursor todavía está posicionado en la fila 1. Sin embargo, si la fila eliminada era la última fila, entonces el cursor está posicionado después de la última fila. Hay que tener en cuenta que no hay un “deshacer” para deleteRow( ), al menos, no por defecto.

- **public boolean rowDeleted( ) throws SQLException**

Cuando un ResultSet puede detectar eliminaciones, el método rowDeleted( ) se utiliza para indicar que se ha eliminado una fila pero permanece como parte del objeto ResultSet. Entonces, si estámos trabajando con un ResultSet que se está pasando entre métodos y compartiendo entre clases, podemos usar rowDeleted( ) para detectar si la fila actual contiene datos válidos.

Actualizando columnas usando objetos

Un aspecto interesante de los métodos getObject( ) y updateObject( ) es que recuperan una columna como un objeto Java. Y porque cada objeto Java puede convertirse en un String utilizando el método toString( ) del objeto, podemos recuperar el valor de cualquier columna en la base de datos e imprimir el valor en la consola como un String.

En el otro sentido, hacia la base de datos, también se pueden usar cadenas para actualizar casi todas las columnas de un ResultSet. Todos los tipos SQL más comunes (integer, float, double, long y date) están envueltos por sus objetos Java representativos: **Integer, Float, Double, Long** y **java.sql.Date.** Cada uno de estos objetos tiene un método valueOf( ) que toma un String.

El método updateObject( ) toma dos argumentos: el primero, un nombre de columna (String) o un índice de columna; y el segundo, un **Object.** Podemos pasar un String como el tipo de Object, y siempre que el String cumpla los requisitos del método valueOf( ) para el tipo de columna, el String será correctamente convertido y almacenado en la base de datos como el tipo SQL deseado.

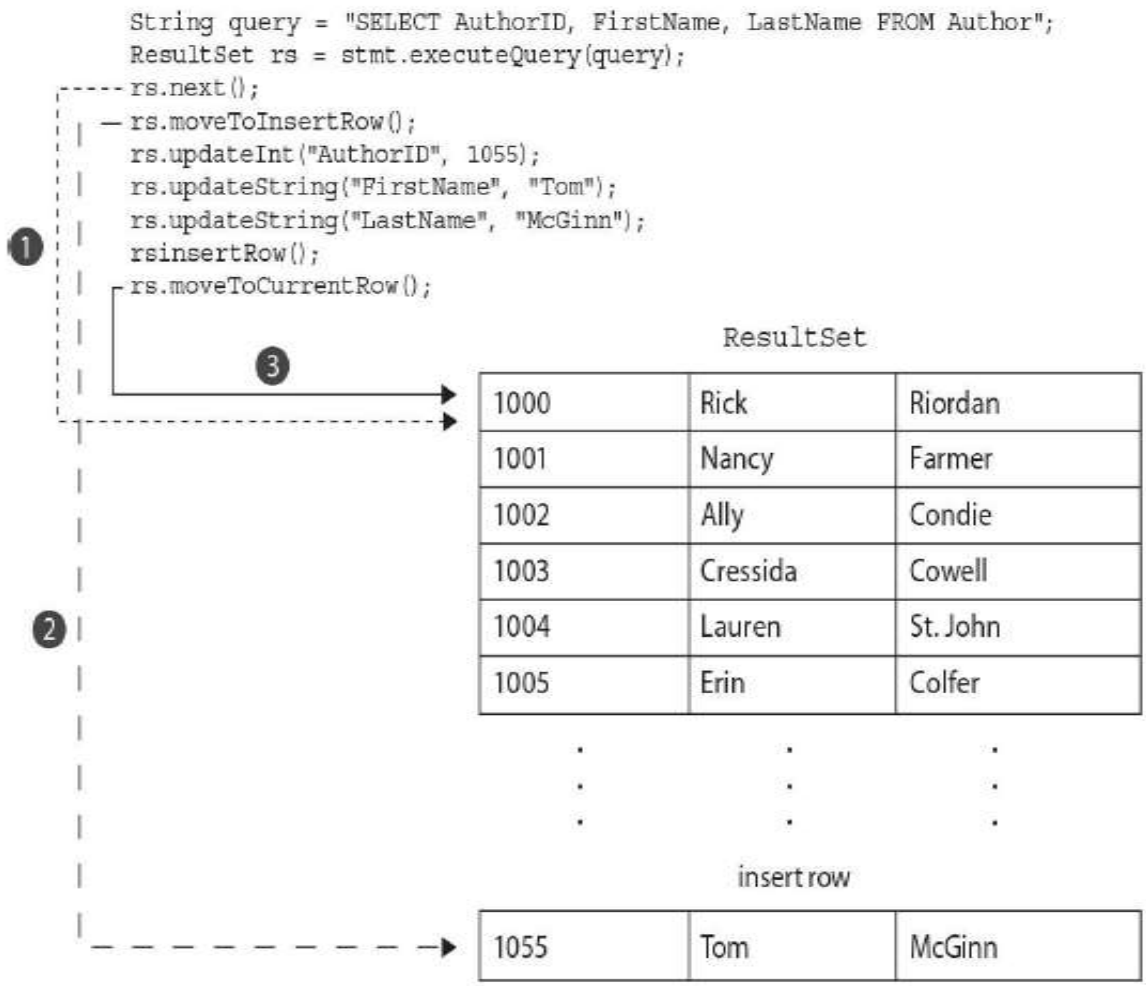
```
// Tenemos una conexión y estamos en un bloque try-catch…
Statement stmt= conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
String query = "SELECT * from Book  WHERE ISBN = '142311339X'";
ResultSet rs = stmt.executeQuery(query);
rs.next( );
rs.updetaObject("PubDate", "2005-04-23");
rs.updateRow( );
```

La cadena que pasamos cumple los requisitos para **java.sql.Date, "aaaa-[m] m- [d] d "**, por lo que la cadena se convierte y almacena correctamente en la base de datos como el valor SQL Date: **2005-04-23**. Hay que tener en cuenta que esta técnica está limitada a aquellos tipos SQL que se pueden convertir a y desde un String, y si el String pasado a el método valueOf( ) para el tipo SQL de la columna no es correctamente formateado para el objeto Java, se lanza una **IllegalArgumentException**.

Insertando nuevas filas usando un ResultSet

ResultSet proporciona una fila especial, llamada fila de inserción, que realmente estamos modificando (actualizando) antes de realizar la inserción. Pensemos en la fila de inserción como un búfer donde podemos modificar una fila vacía de su ResultSet con valores.

Insertar una fila es un proceso de tres pasos, como se muestra en la Figura: Primero (1) mover a la fila de inserción especial, luego (2) actualizar los valores de las columnas para la nueva fila, y finalmente (3) realizar la inserción real (escriba en la base de datos subyacente). El ResultSet existente no cambia; debemos volver a ejecutar nuestra consulta para ver los cambios subyacentes en la base de datos. Sin embargo, podemos insertar tantas filas como queramos. Hay que tener en cuenta que cada uno de estos métodos arroja una SQLException si el tipo de concurrencia del conjunto de resultados se establece en CONCUR_READ_ONLY. Veamos los métodos antes de ver código de ejemplo.



- **public void moveToInsertRow( ) throws SQLException**

Este método mueve el cursor para insertar un búfer de fila. Donde estaba el cursor cuando este método fue llamado es recordado. Después de llamar a este método, los métodos apropiados de actualización son invocados para actualizar los valores de las columnas.

```
rs.moveToInsertRow( );
```

- **public void insertRow( ) throws SQLException**

Este método escribe la inserción del buffer de fila en la base de datos. Tenga en cuenta que el cursor debe estar en la fila de inserción cuando se llama a este método. Además, hay que tener en cuenta que cada columna debe establecerse en un valor antes de que se inserte la fila en la base de datos o se producirá una SQLException. El método insertRow( ) se puede llamar más de una vez; sin embargo, insertRow sigue las mismas reglas que un comando SQL INSERT. A no ser que la clave primaria se genere automáticamente, dos inserciones de los mismos datos darán como resultado SQLException (clave primaria duplicada).

```
rs.insertRow( );
```

- **public void moveToCurrentRow( ) throws SQLException**

Este método devuelve el cursor del conjunto de resultados a la fila donde estaba el cursor antes de llamar al método moveToInsertRow( ).

```
// Tenemos una conexión y estamos en un bloque try-catch…
Statement stmt= conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT AuthorID, FirstName; LastName
                                  FROM Author");
rs.next( );
rs.moveToInsertRow( );       // Mover la fila de inserción especial
rs.updateInt("AuthorID",1055);
rs.updateString("FirstName", "Tom");
rs.updateString("LastName", "McGinn");
rs.insertRow( );             // Insertar la fila en la base de datos
rs.moveToCurrentRow( );    // Mover de vuelta a la fila actual en el ResultSet
```

Obteniendo información sobre la base de datos usando DatabaseMetaData

El objeto **Connection** que obtuvimos de **DriverManager** es un objeto que representa una conexión real con la base de datos. Y mientras que el objeto **Connection** se usa principalmente para crear objetos **Statement**, hay un par de métodos importantes que estudiar en el interfaz **Connection.** Un **Connection** también se puede utilizar para obtener información sobre la base de datos. Estos datos se denominan "metadatos" o "datos sobre datos".
Uno de los métodos de Connection devuelve una instancia a un objeto DatabaseMetaData, a través del cual podemos obtener información sobre la base de datos, sobre el driver, y sobre la semántica de transacciones que la base de datos y el driverr JDBC soporta. Para obtener una instancia de un objeto DatabaseMetaData, utilizamos el método de Connection getMetaData( ):

```
String url = "jsbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
  Connection conn = DriverManager.getConnection(url, user, pwd);
  DatabaseMetaData dbmd = conn.getMetaData( );
} catch(SQLException se){ }
```

Aquí hay algunos métodos que destacaremos:

- **getColumns( )** Devuelve una descripción de columnas en un catálogo especificado y esquema.
- **getProcedures( )** Devuelve una descripción de los procedimientos almacenados en un catálogo y esquema dado.
- **getDriverName( )** Devuelve el nombre del controlador JDBC.
- **getDriverVersion( )** Devuelve el número de versión del controlador JDBC como una string.
- **supportsANSI92EntryLevelSQL( )** Devuelve un verdadero booleano si la base de datos admite la gramática de nivel de entrada ANSI92.

- **public ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern) throws SQLException**

En una base de datos, un **esquema (schema)** es un objeto que impone la integridad de las tablas en la base de datos. El nombre del esquema generalmente es el nombre de la persona que creó la base de datos. En nuestros ejemplos, la base de datos BookGuy contiene la colección de tablas y es el nombre del esquema. Las bases de datos pueden tener múltiples esquemas almacenados en un catálogo.
En este ejemplo, usando la base de datos Java DB como nuestra base de datos de muestra, el catálogo es nulo y nuestro esquema es "BOOKGUY", y estamos usando un patrón coge-todo SQL "%" para los patrones de nombre de tabla y columna, como el "*" al que estamos acostumbrados con sistemas de archivos como Windows. Por lo tanto, nosotros vamos a recuperar todas las tablas y columnas del esquema. Específicamente, vamos a imprimir el nombre de la tabla, el nombre de la columna, el tipo de datos SQL para la columna y el tamaño de la columna.

```
String url = “jsbc:derby://localhost:1521/BookSellerDB”;
String user = “bookguy”;
String pwd = “$3lleR”;
try {
   Connection conn = DriverManager.getConnection(url, user, pwd);
   DatabaseMetaData dbmd = conn.getMetaData( );
   ResultSet rs = dbmd.getColumns(null, “BOOKGUY”, “%”, “%”);
   while(rs.next( )) {
       out.print(“Table Name: “ + rs.getString(“TABLE_NAME”) + “  “);
       out.print(“Table Name: “ + rs.getString(“TABLE_NAME”) + “  “);
       out.print(“Table Name: “ + rs.getString(“TABLE_NAME”) + “  “);
       out.print(“Table Name: “ + rs.getString(“TABLE_NAME”) + “  “);
   }
 }catch(SQLException se){  out.println(out.println(“SQLException: ” +se); }
```

- **public ResultSet getProcedures(String catalog, String schemaPattern, String procedureNamePattern) throws SQLException**

Este método devuelve un conjunto de resultados que contiene información descriptiva sobre los procedimientos almacenados para un catálogo y esquema. Este ejemplo devuelve el nombre de cada procedimiento almacenado en la base de datos.

```
try {
  Connection con = …
  DatabaseMetaData dbmd = conn.getMetaData( );
  ResultSet rs = dbmd.getProcedures(null,null, “%”);
  while(rs.next( )) {
    out.println(“Procedure Name: ” + rs.getString(“PROCEDURE_NAME”));
  }
} catch(SQLException se) { }
```

- **public String getDriverName( ) throws SQLException**

Este método simplemente devuelve el nombre del driverJDBC como una string.

```
 System.out.println(“getDriverName: ” + dbmd.getDriverName());
```

- **public String getDriverVersion( ) throws SQLException**

Este método devuelve el número de versión del controlador JDBC como una cadena.

```
Logger logger = Logger.getLogger(“com.cert.DatabaseMetaDataTest”);
Connection conn = …
DatabaseMetaData dbmd = conn.getMetaData( );
logger.log(Level.INFO, “Driver Version: {0}”, dbmd.getDriverVersion( ));
logger.log(Level.INFO, “Driver Name: {0}”, dbmd.getDriverName( ));
```

- **public Boolean supportsANSI92EntryLevelSQL( ) throws SQLException**

Este método devuelve true si la base de datos y el driver JDBC admiten la gramática de nivel de entrada ANSI SQL-92. Un mínimo de soporte para este nivel es un requisito para los drivers JDBC (y, por lo tanto, la base de datos).

```
Connection conn = …
DatabaseMetaData dbmd = conn.getMetaData( );
if (!dbmd.supportsANSI92EntryLevelSQL( )) {
   logger.log(Level.WARNING, “JDBC Driver does not meet minimum requirements for
                  SQL-92 support);
 }
```

Cuando las cosas van mal: Excepciones y Avisos

Al igual que otras excepciones de Java, SQLException es una forma para que nuestra aplicación determine cuál es el problema y tome medidas si es necesario. Veamos el tipo de datos que obtiene de una excepción SQLE a través de sus métodos.

- **public String getMessage( )**

Este método en realidad se hereda de **java.lang.Exception**, la cual extiende **SQLException**. Este método devuelve la razón detallada por la que se lanzó la excepción. A menudo, el mensaje contiene un SQLState y el código de error que proporcionan información específica sobre qué salió mal.

- **public String getSQLState( )**

El String devuelto por getSQLState proporciona un código específico y un mensaje relacionado. Los mensajes SQLState están definidos por los **estándares X / Open y SQL: 2003;** sin embargo, depende de implementación para usar estos valores. Podemos determinar qué estándar utiliza el controlador JDBC(o si no lo hace) a través de método **DatabaseMetaData.getSQLStateType( ).**

- **public int getErrorCode( )**

Los códigos de error no están definidos por un estándar y son, por lo tanto, específicos de implementación.

- **public SQLException getNextException( )**

Uno de los aspectos interesantes de SQLException es que la excepción lanzada podría ser el resultado de más de un problema. Afortunadamente, JDBC simplemente agrega cada excepción a la siguiente en un proceso llamado encadenamiento. Típicamente, la excepción más severa es lanzada la última, de forma que es la primera excepción en la cadena. Podemos obtener una lista de todas las excepciones en la cadena utilizando el método getNextException( ) para recorrer en iteración la lista. Cuando se alcanza el final de la lista, getNextException( ) devuelve un valor nulo.

```
Logger logger = Logger.getLogger("com.example.MyClass");
try{      // Algo de código JDBC …
}catch ( SQLException se) {
  while(se != null) {
    logger.log(Level.SEVERE, "------ SQLException -------");
    logger.log(Level.SEVERE,"SQLState:  " + se.getSQLState( ));
    logger.log(Level.SEVERE, "Vendor Error code:  "+ se.getErrorCode( ));
    logger.log(Level.SEVERE, "Message: " + se.getMessage( ));
    se = se.getNextException( );
  }
}
```

Advertencias (Warnings)

Aunque **SQLWarning** es una subclase de **SQLException,** las advertencias son silenciosamente encadenadas al objeto JDBC que las reportó. Esta es probablemente una de los pocas veces en Java donde un objeto que forma parte de una jerarquía de excepción no es lanzado como una excepción. La razón es que una advertencia no es una excepción de por sí. Las advertencias se pueden reportar en objetos **Connection, Statement** y **ResultSet.**

Por ejemplo, supongamos que configuramos por error el tipo de conjunto de resultados en **TYPE_SCROLL_SENSITIVE** al crear un **Statement**. Esto no crea una excepción; en cambio, la base de datos manejará la situación encadenando un **SQLWarning** al objeto **Connection** y restableciendo el tipo a **TYPE_FORWARD_ONLY** (el valor predeterminado) y continúa. Todo estaría bien, por supuesto, hasta que intentemos colocar el cursor, en ese punto se lanzaría una **SQLException**. Y, al igual que **SQLException**, podemos recuperar advertencias del objeto **SQLWarning** utilizando el método getNextWarning( ).

```
Connection con =
DriverManager.getConnection("jdbc:derby://localhost:1527/BookSellerDB",
 "bookguy", "$3lleR");
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
String query = "SELECT * from Book WHERE Book.Format ='Hardcover' ";
ResultSet rs = stmt.executeQuery(query);
SQLWarning warn = conn.getWarnings( );
while(warn != null) {
  out.println("SQLState: " + warn.getSQLState( ));
  out.println("Message: " + warn.getMessage( ));
  warn = warn.getNextWarning( );
}
```

Los objetos **Connection** agregarán advertencias (si es necesario) hasta que la conexión este cerrada o hasta que se llame al método clearWarnings( ) en la instancia de **Connection**. El método clearWarnings( ) establece la lista de advertencias en nulo hasta que se informa otra advertencia para este objeto **Connection**.
Las declaraciones y los ResultSets también generan SQLWarnings y estos objetos tienen sus propios métodos clearWarnings( ). Las advertencias de Statement se borran automáticamente cuando una declaración se vuelve a ejecutar y las advertencias **ResultSet** son borradas cada vez que se lee una nueva fila del conjunto de resultados.
A continuación se resumen los métodos asociados con **SQLWarnings**.

- **SQLWarning getWarnings( ) throws SQLException**

Este método obtiene el primer objeto **SQLWarning** o devuelve nulo si no hay advertencias para este objeto **Connection, Statement o ResultSet**. Se lanza una **SQLException** si el método se llama en un objeto cerrado.

- **void clearWarnings( ) throws SQLException**

Este método borra y restablece el conjunto actual de advertencias para este objeto **Connection, Statement o ResultSet**. Se produce una excepción **SQLException** si se llama al método en un objeto cerrado.

Cerrando los recursos SQL de forma adecuada

Es importante saber cuándo un recurso se cierra automáticamente. Cada de los tres objetos principales JDBC (**Connection, Statement y ResultSet**) tiene un método close( ) para cerrar explícitamente el recurso asociado con el objeto y liberar explícitamente el recurso. Los objetos tienen una relación entre sí, por lo que si un objeto ejecuta close( ), tendrá un impacto en los otros objetos.

Method Call	Has the Following Action(s)
`Connection.close()`	Releases the connection to the database. Closes any `Statement` created from this `Connection`.
`Statement.close()`	Releases this `Statement` resource. Closes any open `ResultSet` associated with this `Statement`.
`ResultSet.close()`	Releases this `ResultSet` resource. Note that any `ResultSetMetaData` objects created from the `ResultSet` are still accessible.
`Statement.executeXXXX()`	Any `ResultSet` associated with a previous `Statement` execution is automatically closed.

Es una buena práctica minimizar la cantidad de veces que cerramos y recreamos objetos **Connection.** Como regla general, crear la conexión con la base de datos y pasar las credenciales de nombre de usuario y contraseña para la autenticación es un proceso relativamente costoso, por lo que realizar la actividad una vez por cada consulta SQL puede hacer que el código se ejecute lentamente. De hecho, normalmente, las conexiones a la base de datos se crean en un grupo e instancias de conexión y se entregan a las aplicaciones según sea necesario, en lugar de permitir o requerir aplicaciones individuales para crearlos.
Los objetos **Statement** son menos costosos de crear. Hay maneras de precompilar sentencias SQL utilizando un **PreparedStatement**, que reduce el gasto de recursos asociados con la creación de cadenas de consulta SQL y el envío de esas cadenas a la base de datos para su ejecución.
Los **ResultSets** son los objetos menos costosos para crear, y como vimos en la sección "ResultSets", para obtener resultados de una sola tabla, podemos usar el ResultSet para actualizar, insertar y eliminar filas, por lo que puede ser muy eficiente de usar un **Resultset.**

```
Connection con = null;
String url, user, pwd;
try {
   conn = DriverManager.getConnection(url, user, pwd);
   Statement stmt = conn.createStatement( );
   ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
    // … procesamos los resultados
    // …
    if ( rs != null && stmt ¡= null) {
       rs.close( );
       stmt.close( );
   }
 }catch(SQLException se) {
    out.println("SQLException: " + se);
 } finally {
      try {
        if(conn != null) {
           conn.close( );
        }
      } catch(SQLException sec) {
   out.println("Exception closing connection!");
  }
}
```

Observemos todo el trabajo que tenemos que realizar para cerrar la conexión: Primero debemos asegurarnos de que realmente tenemos un objeto y no un valor nulo, y luego necesitamos probar el método close( ) dentro de otro **try** dentro del bloque **finally.** Afortunadamente, hay una manera más fácil...

Usando try-with-resources para cerrar Connections, Statements y ResultSets

La declaración de try-with-resources llamará automáticamente al método close( ) en cualquier recurso declarado en el paréntesis al final del bloque try.

```
String url, user, pwd;
try (Connection conn = DriverManager.getConnection(url, user, pwd)) {
   Statement stmt = conn.createStatement( );
   ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
    // … procesamos los resultados
    if ( rs != null && stmt ¡= null) {
       rs.close( );
       stmt.close( );
   }
  }catch(SQLException se) {
    out.println("SQLException: " + se);
 }
```

Un recurso declarado en la declaración try-with-resource debe implementar la interfaz **AutoCloseable.** Uno de los cambios para JDBC en Java SE 7 (JDBC 4.1) fue la modificación de la API para que **Connection, Statement** y **ResultSet** extiendan la interfaz **AutoCloseable** y soporten la gestión automática de recursos.

Hay que tener en cuenta que debemos incluir el tipo de objeto en la declaración dentro del paréntesis Lo siguiente provocará un error de compilación:

```
try (conn = DriverManager.getConnection(url, user, pwd)) {
```

Con try-with-resources también se puede usar con múltiples recursos, por lo que también podríamos incluir la declaración de Statement en el try:

```
try (Connection conn = DriverManager.getConnection(url, user, pwd);
      Statement stmt = conn.createStatement( )) {
```

Hay que tener en cuenta que cuando se declara más de un recurso en try-with-resources, los recursos serán cerrados en el orden inverso de su declaración (entonces stmt.close( ) se llamará primero, seguido de conn.close( )).

En un try-with-resources cualquier excepción lanzada como resultado del cierre de recursos al final del bloque try es suprimida, si también hubo una excepción lanzada en el bloque try. Estas excepciones se pueden recuperar de la excepción lanzada llamando al método getSuppressed( ) en la excepción lanzada.

```
} catch (SQLException se) {
   out.println("SQLException: " + se);
   Throwable[ ] supressed = se.getSuppresed( );    // Recoge un array de excepciones
                                                   // suprimidas
   for(Throwable t : suppressed) {  // Iteramos por el array
       out.println("Suppressed exception: " + t);
   }
}
```