JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps you to write java applications that manage these three programming activities:

- Connect to a data source, like a database
- Send queries and update statements to the database
- Retrieve and process the results received from the database in answer to your query

JDBC Product Components

JDBC includes four components:

1. **The JDBC API —**

   The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

   The JDBC API is part of the Java platform, which includes the *Java™ Standard Edition* (Java™ SE) and the *Java™ Enterprise Edition* (Java™ EE). The JDBC 4.0 API is divided into two packages: `java.sql` and `javax.sql`. Both packages are included in the Java SE and Java EE platforms.
2. **JDBC Driver Manager —**
   The JDBC `DriverManager` class defines objects which can connect Java applications to a JDBC driver.

`DriverManager` has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

The Standard Extension packages `javax.naming` and `javax.sql` let you use a `DataSource` object registered with a *Java Naming and Directory Interface*™ (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism.

3. **JDBC Test Suite —**
   The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

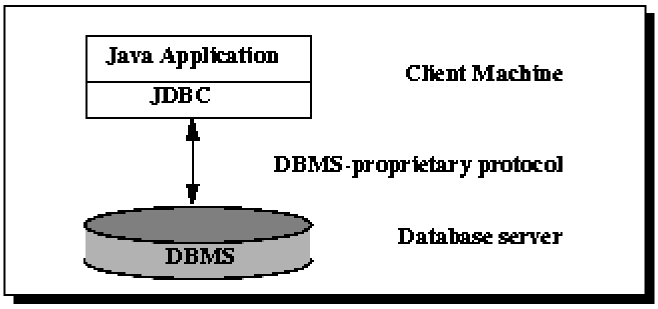
4. **JDBC-ODBC Bridge —**
   The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

JDBC Architecture

Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

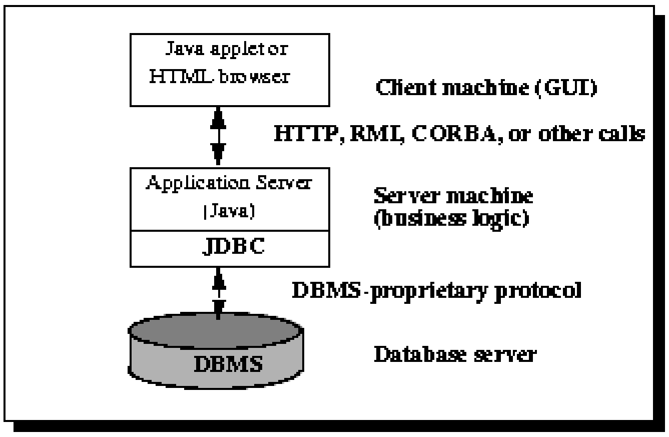
Figure 1: Two-tier Architecture for Data Access.



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 2: Three-tier Architecture for Data Access.


Java applet or HTML browser
Client machine (GUI)
HTTP, RMI, CORBA, or other calls
Application Server (Java)
JDBC
Server machine (business logic)
DBMS-proprietary protocol
DBMS
Database server

Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java byte code into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected row sets. The JDBC API is also what allows access to a data source from a Java middle tier.

JDBC drivers.

A JDBC driver can come from many sources: database software, such as Java DB, a JDBC driver vendor such as DataDirect, Oracle, MySQL, or an ISV/OEM such as Sun.

Your driver should include instructions for installing it. For a JDBC driver written for specific Database Management Systems (DBMS), installation consists of copying the driver onto your machine. No special configuration is needed.

Types of Drivers

There are many possible implementations of JDBC drivers. These implementations are categorized as follows:

- Type 1 - drivers that implement the JDBC API as a mapping to another data access API, such as ODBC. Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a Type 1 driver.
- Type 2 - drivers that are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited.
- Type 3 - drivers that use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client’s requests to the data source.
- Type 4 - drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source

The classes that we use for JDBC programming are contained in the **java.sql and javax.sql packages**

java.sql Package

The key interfaces in the JDBC Core API are as follows:

1) **java.sql.Driver / java.sql.DriverManager** The Driver object implements the acceptsURL(String url)method, confirming its ability to connect to the URL the DriverManager passes.Then the connect() method of the Driver class is automatically invoked when user makes a call to getConnection() method of the DriverManger Class. It returns a connection object on successful connection else it returns null .

   static Connection getConnection(String url,String user, String password) establishes a connection to the given database and returns a Connection object.

   Parameters: 1) url- the URL for the database

2) user- the database login ID

3) password -the database login password

2) **java.sql.Connection.** The Connection object provides the connection between the JDBC API and the database management system the URL specifies. A Connection represents a session with a specific database.

- **Statement createStatement()**
  - creates a statement object that can be used to execute SQL queries and updates without parameters.
- **void close()**
  - immediately closes the current connection.

3) **java.sql.Statement.** The Statement object acts as a container for executing a SQL statement on a given Connection.

- **ResultSet executeQuery(String sql)**
  - executes the SQL statement given in the string and returns a ResultSet to view the query result.
  - Parameters: sql the SQL query
- **int executeUpdate(String sql)**
  - executes the SQL INSERT, UPDATE, or DELETE statement specified by the string. Also used to execute Data Definition Language (DDL) statements such as CREATE TABLE.Returns the number of records affected, or -1 for a statement without an update count.
  - Parameters: sql the SQL statement
- **Boolean execute(String sql)**

    - executes the SQL statement specified by the string. Returns true if the statement returns a result set, false otherwise. Use the getResultSet or getUpdateCount method to obtain the statement outcome.
    - Parameters: sql the SQL statement
- **int getUpdateCount()**
    - Returns the number of records affected by the preceding updatestatement, or -1 if the preceding statement was a statement without an update count. Call this method only once per executed statement.
- **ResultSet getResultSet()**
    - Returns the result set of the preceding query statement, or null if the preceding statement did not have a result set. Call this method only once per executed statement.

4) java.sql.ResultSet. The ResultSet object controls access to the results of a given Statement in a structure that can be traversed by moving a cursor and from which data can be accessed using a family of getter methods.

- **Boolean next()**

makes the current row in the result set move forward by one. Returns false after the last row. Note that you must call this method to advance to the first row.

- **Xxx getXxx(int columnNumber)**

(coloumn no starts from 1 for the first coloumn)

- **Xxx getXxx(String columnName)**

(Xxx is a type such as int, double, String, Date, etc.)

return the value of the column with column index columnNumber or with column names, converted to the specified type. Not all type conversions are legal.

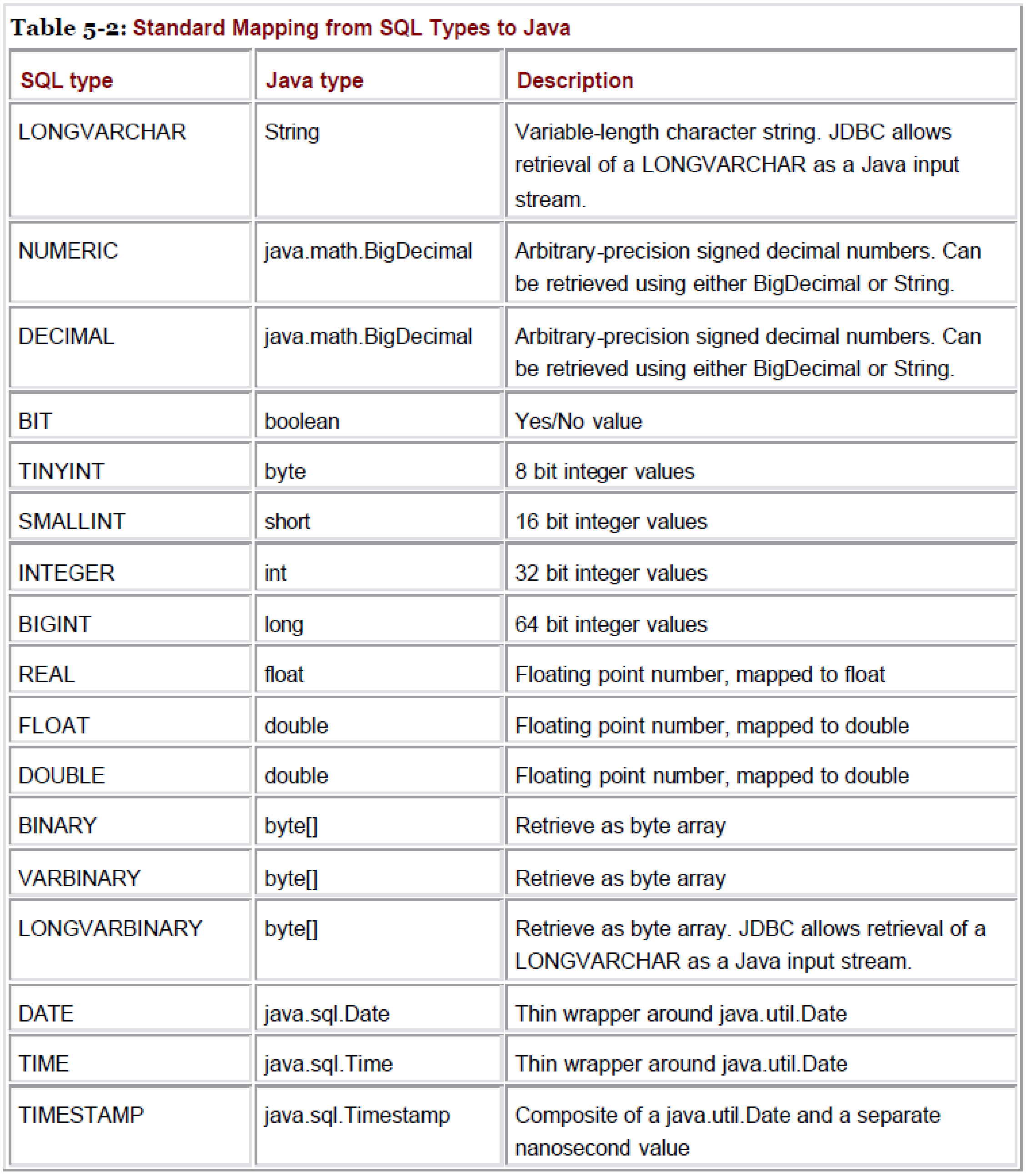


Table 5-2: Standard Mapping from SQL Types to Java

SQL type	Java type	Description
LONGVARCHAR	String	Variable-length character string. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
NUMERIC	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
DECIMAL	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
BIT	boolean	Yes/No value
TINYINT	byte	8 bit integer values
SMALLINT	short	16 bit integer values
INTEGER	int	32 bit integer values
BIGINT	long	64 bit integer values
REAL	float	Floating point number, mapped to float
FLOAT	double	Floating point number, mapped to double
DOUBLE	double	Floating point number, mapped to double
BINARY	byte[]	Retrieve as byte array
VARBINARY	byte[]	Retrieve as byte array
LONGVARBINARY	byte[]	Retrieve as byte array. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
DATE	java.sql.Date	Thin wrapper around java.util.Date
TIME	java.sql.Time	Thin wrapper around java.util.Date
TIMESTAMP	java.sql.Timestamp	Composite of a java.util.Date and a separate nanosecond value

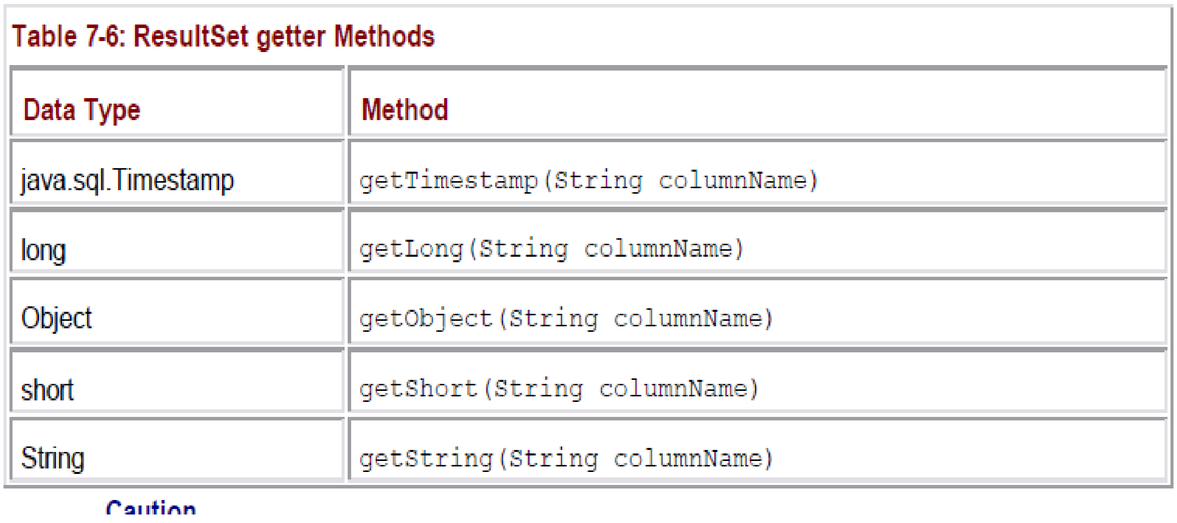


Table 7-6: ResultSet getter Methods

Data Type	Method
java.sql.Timestamp	`getTimestamp(String columnName)`
long	`getLong(String columnName)`
Object	`getObject(String columnName)`
short	`getShort(String columnName)`
String	`getString(String columnName)`

Caution

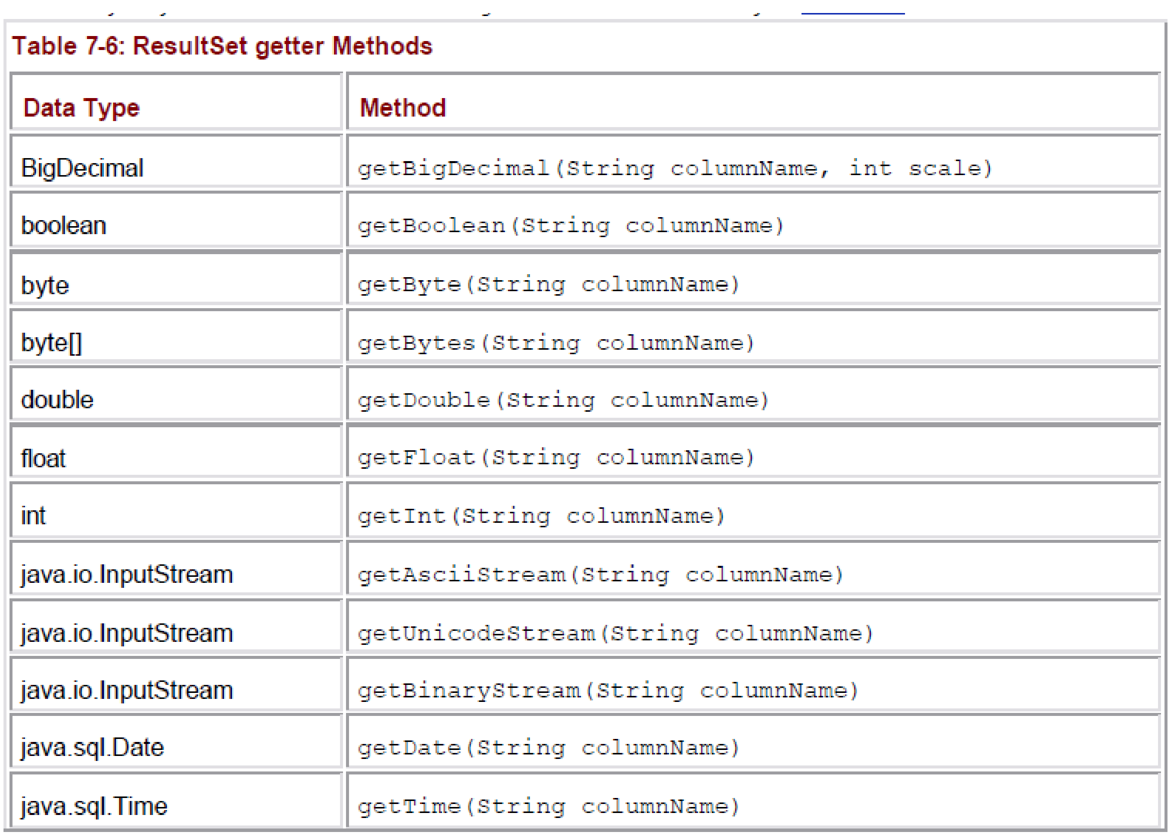


Table 7-6: ResultSet getter Methods

Data Type	Method
BigDecimal	`getBigDecimal(String columnName, int scale)`
boolean	`getBoolean(String columnName)`
byte	`getByte(String columnName)`
byte[]	`getBytes(String columnName)`
double	`getDouble(String columnName)`
float	`getFloat(String columnName)`
int	`getInt(String columnName)`
java.io.InputStream	`getAsciiStream(String columnName)`
java.io.InputStream	`getUnicodeStream(String columnName)`
java.io.InputStream	`getBinaryStream(String columnName)`
java.sql.Date	`getDate(String columnName)`
java.sql.Time	`getTime(String columnName)`

- **int findColumn(String columnName)**

gives the column index associated with a column name.

- **void close()**

immediately closes the current result set.

There are three exceptions included in the package namely

a) **SQLException** : An exception that provides information on a database access error or other errors. Each SQLException provides several kinds of information:

- a string describing the error. This is used as the Java Exception message, available via the method getMesage.
- a "SQLstate" string, which follows either the XOPEN SQLstate conventions or the SQL 99 conventions. The values of the SQLState string are described in the appropriate spec. The DatabaseMetaData method getSQLStateType can be used to discover whether the driver returns the XOPEN type or the SQL 99 type.
- an integer error code that is specific to each vendor. Normally this will be the actual error code returned by the underlying database.
- a chain to a next Exception. This can be used to provide additional error information.

b) **BatchUpdateException**

An exception thrown when an error occurs during a batch update operation. In addition to the information provided by

SQLException, a BatchUpdateException provides the update counts for all commands that were executed successfully during the batch update, that is, all commands that were executed before the error occurred. The order of elements in an array of update counts corresponds to the order in which commands were added to the batch.

c) **SQLWarning :** An exception that provides information on database access warnings. Warnings are silently chained to the object whose method caused it to be reported. Warnings may be retrieved from Connection, Statement, and ResultSet objects. Trying to retrieve a warning on a connection after it has been closed will cause an exception to be thrown. Similarly, trying to retrieve a warning on a statement after it has been closed or on a result set after it has been closed will cause an exception to be thrown. Note that closing a statement also closes a result set that it might have produced.

d) **Data Truncation** : An exception that reports a DataTruncation warning (on reads) or throws a DataTruncation exception (on writes) when JDBC unexpectedly truncates a data value.

There are seven basic steps to using JDBC to access a database.

They are

1. Import the java.sql package
2. Load and register the driver.
3. Establish a connection to the database driver.
4. Create a statement.
5. Execute the statement.
6. Retrieve the results.
7. Close the statement and connection.

An example jdbc program

```
import java.sql.*; // imports the JDBC core package
public class JdbcDemo{
public static void main(String args[]){
int qty;
float cost;
String name;
String desc;
// SQL Query string
String query = "SELECT * from stock";
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// load the JDBC driver
Connection con = DriverManager.getConnection
("jdbc:odbc:inventory");
// get a connection to jdbcodbc driver and access dsn
inventory
Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query); // execute query

while (rs.next()) { // parse the results

name = rs.getString("stckid");

desc = rs.getString("stckname");

qty = rs.getInt("stckamt");

System.out.println(name+", "+desc+"\t: "+qty);

}

con.close();

}

catch(ClassNotFoundException e){

e.printStackTrace();

}


catch(SQLException e){

e.printStackTrace();

}
```

}}

The DriverManager Class

The java.sql.DriverManager provides basic services for managing JDBC drivers. During initialization, the DriverManager attempts to load the driver classes referenced in the "jdbc.drivers" system property. Alternatively, a program can explicitly load JDBC drivers at any time using Class.forName(). This allows a user to customize theJDBC drivers their applications use.

A newly loaded driver class should call registerDriver() to make itself known to the DriverManager. Usually, the driver does this internally.

When getConnection() is called, the DriverManager attempts to locate a suitable driver from among those loaded at initialization and those loaded explicitly using the same class loader as the current applet or application. It does this by polling all registered drivers, passing the URL of the database to the drivers' acceptsURL()method.

There are three forms of the getConnection() method, allowing the user to pass additional arguments in addition to the URL of the database:

a) public static synchronized Connection getConnection(String url) throwsSQLException
b) public static synchronized Connection getConnection(String url,String user,String password)throws SQLException
c) public static synchronized Connection getConnection(String url,Properties info)throwsSQLException

Note: When searching for a driver, JDBC uses the first driver it finds that can successfully connect to the given URL. It starts with the drivers specified in the sql.drivers list, in the order given. It then tries the loaded drivers in the order in which they are loaded.

JDBC(Database) URLs

A URL (Uniform Resource Locator) is an identifier for locating a resource on the Internet. It can be thought of as an address. A JDBC URL is a flexible way of identifying a database so that the appropriate driver recognizes it and establishes a connection with it. JDBC URLs allow different drivers to use different schemes for naming databases. The odbc sub protocol, for example, lets the URL contain attribute values.

The standard syntax for JDBC URLs is shown here:

jdbc:<subprotocol>:<subname>

The three parts of a JDBC URL are broken down as follows:

a) jdbc — The protocol. The protocol in a JDBC URL is always jdbc.

b) <subprotocol> — The name of the driver or connectivity mechanism, which may be supported by one or more drivers

c) <subname> — A unique identifier for the database

For example, this is the URL to access the contacts database through theJDBC-ODBC bridge:

jdbc:odbc:contacts

The odbc subprotocol has the special feature of allowing any number of attribute values to be specified after the database name, as shown here:

jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]*

Attributes passed in this way may include user id and password, for example.

An eg URL for Oracle driver

jdbc:Oracle:thin:@server:1521:Oracle8i

Connection Interface

A Connection object represents a connection with a database. A connection session includes the SQL statements that are executed and the results that are returned over that connection. A single application can have one or more connections with a singledatabase, or it can have connections with many different databases.

Opening a connection

The standard way to establish a connection with a database is to call the method getConnection() on either a DataSource or a DriverManager. The Driver method connect() uses this URL to establish the connection.

A user can bypass the JDBC management layer and call Driver methods directly. This can be useful in the rare case that two drivers can connect to a database and the user wants explicitly to select a particular driver. Usually, however, it is much easier to just let the DataSource class or the DriverManager class open a connection.

Having established a connection to the database, you are now in a position to execute a SQL statement.

SQL Statements

Once a connection is established, it is used to pass SQL statements to the database. Since there are no restrictions imposed on the kinds of SQL statements that may be sent to a DBMS using JDBC, the user has a great deal of flexibility to use database-specific statements or even non-SQL statements.

Statement Interface

The JDBC core API provides these three classes for sending SQL statements to the database:

1. Statement. A Statement object is used for sending simple SQL statements. Statements are created by the method createStatement() via a connection object.

2. PreparedStatement. A PreparedStatement is a SQL statement that is precompiled and stored in a PreparedStatement object. This object can then be used to execute this statement multiple times.They are created by the method prepareStatement() via a connection object.
3. CallableStatement. CallableStatement is used to execute SQL stored procedures. CallableStatements are created by the method prepareCall() via connection object.

Statement object

A Statement object is used for executing a static SQL statement and obtaining the results it produces. Statement defines these three methods for executing SQL statements, which handle SQL commands returning different kinds of results:

1. executeUpdate(String sql): Execute a SQL INSERT, UPDATE, or DELETE statement, which returns either a count of rows affected or zero.
2. executeQuery(String sql): Execute a SQL statement that returns a single ResultSet.
3. execute(String sql): Execute a SQL statement that may return multiple results.

The executeUpdate() method is used for SQL commands such as INSERT, UPDATE,and DELETE, which return a count of rows affected rather than a ResultSet; or for DDL commands such as CREATE TABLE, which returns nothing, in which case the return value is zero.

The executeQuery() method is used for SQL queries returning a single ResultSet.

The execute method is used to execute a SQL statement that may return multiple results. In some situations, a single SQL statement may return multiple ResultSets and/or update counts. The execute() method returns boolean true if the SQL statement returns a ResultSet and false if the return is an update count.

The Statement object defines the following supporting methods:

- getMoreResults()
- getResultSet()
- getUpdateCount()

These methods let you navigate through multiple results. You can use getResultSet() or getUpdateCount() to retrieve the result and getMoreResults() to move to any subsequent results.

PreparedStatement Object

PreparedStatements are nothing more than statements that are precompiled. Precompilation means that these statements can be executed more efficiently than simple statements, particularly in situations where a Statement is executed repeatedly in a loop.

PreparedStatements can contain placeholders for variables known as IN parameters, which are set using setter methods. The placeholder is represented by a '?' in the query string .Each place holder is numbered starting from 1.

A typical setter method looks like this:

public void setXXX(int parameterIndex, XXX x) throws SQLException where XXX can be any primitive data types like int ,float etc or objects like String,Date,Time etc. and the parameter index is the position number of the place holder that begins with 1.

The following is the list of all the set methods in the primitive data type category:

setBoolean()	setByte()	setInt()	setLong()
setFloat()	setDouble()	setNull()	setShort()

The following is the list of set methods for object parameters:

setString() setBignum() setBytes()

setDate() setTime() setObject()

setTimeStamp()

An example program using prepared statements .In this the following line, which pstmt.setInt(1,2) sets integer parameter #1(the first placeholder) equal to 2:

Using a PreparedStatement

```
Import java.sql.*;
public class PreparedStmt{
public static void main(String args[]){
int qty;
float cost;
String name;
String desc;
String query = ("SELECT * FROM Stock WHERE
Item_Number= ?";
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection con =

DriverManager.getConnection ("jdbc:odbc:Inventory");

PreparedStatement pstmt = con.prepareStatement(query);

pstmt.setInt(1, 2);//set the first place holder as value 2

ResultSet rs = pstmt.executeQuery();

while (rs.next()) {

name = rs.getString("Name");

desc = rs.getString("Description");

qty = rs.getInt("Qty");

cost = rs.getFloat("Cost");

System.out.println(name+",     "+desc+"\t:     "+qty+"\t@

$"+cost);

}

}

catch(ClassNotFoundException e){

e.printStackTrace();
```

```
}
catch(SQLException e){
e.printStackTrace();
}
}
}
```

The methods setBytes and setString are capable of sending unlimited amounts of data. You can also handle large amounts of data by setting an IN parameter to a Java
input stream(for images).

JDBC provides these three methods for setting IN parameters to input streams:

- setBinaryStream(for streams containing uninterpreted bytes)
- setAsciiStream (for streams containing ASCII characters)

- setUnicodeStream (for streams containing Unicode characters)

When the statement is executed, the JDBC driver makes repeated calls to the inputstream, reading its contents and sending them to the database as the actual
parameter value.

The setNull() method allows you to send a NULL value to the database as an IN parameter. You can also send a NULL to the database by passing a Java null value to a setXXX()method.

The setObject() method has three forms.In its simplest form,setObject() takes two parameters.The first is the position of the place holderand the second is the Object.Before sending the Object to the database the driver converts the Object to the standard SQL data type for that object. The second form of the setObject() method adds another input parameter. In this the third parameter represents a datatype to which explicitly convert the Object to.

CallableStatement Object

The CallableStatement object allows you to call a database stored procedure from a Java application. A CallableStatement object contains a call to a stored procedure; it does not contain the stored procedure itself, as the stored procedure is stored in the database. In the below example, we create and use a stored procedure.

```
import java.sql.*;
public class CallableStmt{
public static void main(String args[]){
String name;
String storedProc = "create proc SHOWNAME  as  select * from stock ";
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
    Connection con = DriverManager.getConnection
("jdbc:odbc:inventory");
    Statement stmt = con.createStatement();
    stmt.executeUpdate(storedProc);
    CallableStatement cs = con.prepareCall("{call
SHOWNAME}");
    ResultSet rs = cs.executeQuery();
    while (rs.next()) {
    name = rs.getString("stckname");
    System.out.println(name);
    }
    }
    catch(ClassNotFoundException e){
    e.printStackTrace();
    }
```

Scrollable and Updatable Result Sets

The next `()` method of the `ResultSet` class iterates over
the rows in a result set. That is certainly adequate for a program that needs
to analyze the data. But what if the user had to move both forward and backwards

in the resultset. Then we find that the resultset created must have scrollable

capability.

Furthermore, once you display the contents of a result set to users, they
may be tempted to edit it.

To obtain scrolling result sets from your queries, you must obtain a different
`Statement` object with the method

```
Statement stat = conn.createStatement(type,
concurrency);
```

For a prepared statement, use the call

```
PreparedStatement stat =
conn.prepareStatement(query,type, concurrency);
```

The possible values of `type` and `concurrency` are listed as below

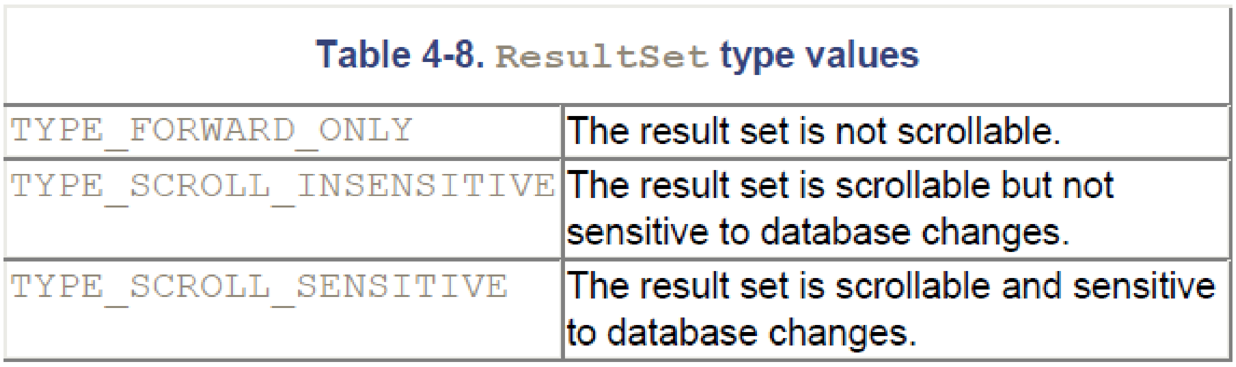


Table 4-8. `ResultSet` type values

`TYPE_FORWARD_ONLY`	The result set is not scrollable.
`TYPE_SCROLL_INSENSITIVE`	The result set is scrollable but not sensitive to database changes.
`TYPE_SCROLL_SENSITIVE`	The result set is scrollable and sensitive to database changes.

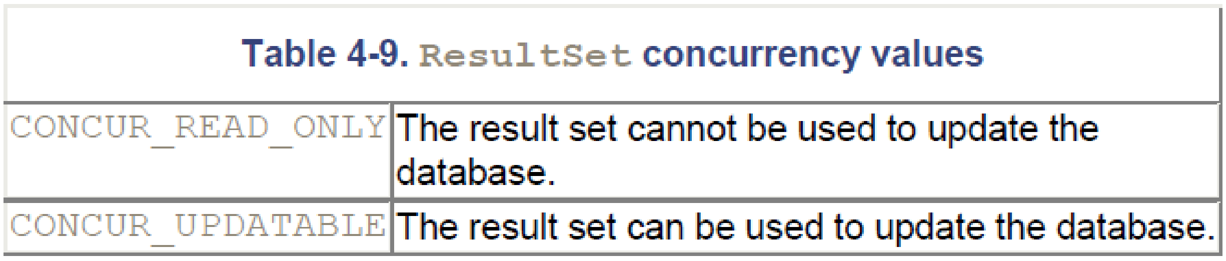


Table 4-9. `ResultSet` concurrency values

`CONCUR_READ_ONLY`	The result set cannot be used to update the database.
`CONCUR_UPDATABLE`	The result set can be used to update the database.

Methods in ResultSet interface for cursor positioning and getting the current cursor position are as follows:

• **boolean previous()**

(JDBC 2) moves the cursor to the preceding row. Returns true if the cursor is positioned on a row.

• **int getRow()**

(JDBC 2) gets the number of the current row. Rows are numbered starting with 1.

• **boolean absolute(int r)**
(JDBC 2) moves the cursor to row r. Returns true if the cursor is positioned on a row.

• **boolean relative(int d)**

(JDBC 2) moves the cursor by d rows. If d is negative, the cursor is moved backward. Returns true if the cursor is positioned on a row.

• **boolean first()**
• **boolean last()**
(JDBC 2) move the cursor to the first or last row. Return true if the cursor is positioned on a row.

• **void beforeFirst()**
• **void afterLast()**
(JDBC 2) move the cursor before the first or after the last row.

• **boolean isFirst()**
• **boolean isLast()**
(JDBC 2) test if the cursor is at the first or last row.

• **boolean isBeforeFirst()**
• **Boolean isAfterLast()**
(JDBC 2) test if the cursor is before the first or after the last row.

Updating a ResultSet

To appreciate the simplicity of using Updatable ResultSet instead of SQL UPDATES, it is worth looking first at what is involved in using Statement.executeUpdate() to change a customer address.
The code to make this change looks like this:

```
stmt.executeUpdate(
"UPDATE Customers SET Street = '123 Main Street' +
"WHERE First_Name = 'Vito' AND Last_Name = 'Corleone'");
```

This is simple enough when you know how to identify the record to be updated, but consider how much more complicated it would be if your application were displaying the ResultSet in a JTable. Unless you go to considerable trouble to keep track of the current record, it is quite difficult to identify to the RDBMS which record to update.
Using an Updatable ResultSet simplifies the situation considerably. All you need to do is set the cursor to the desired row and change the column value using a data-type-specific update method. Here's an example:

```
rs.updateString("Street", "123 Main");
```

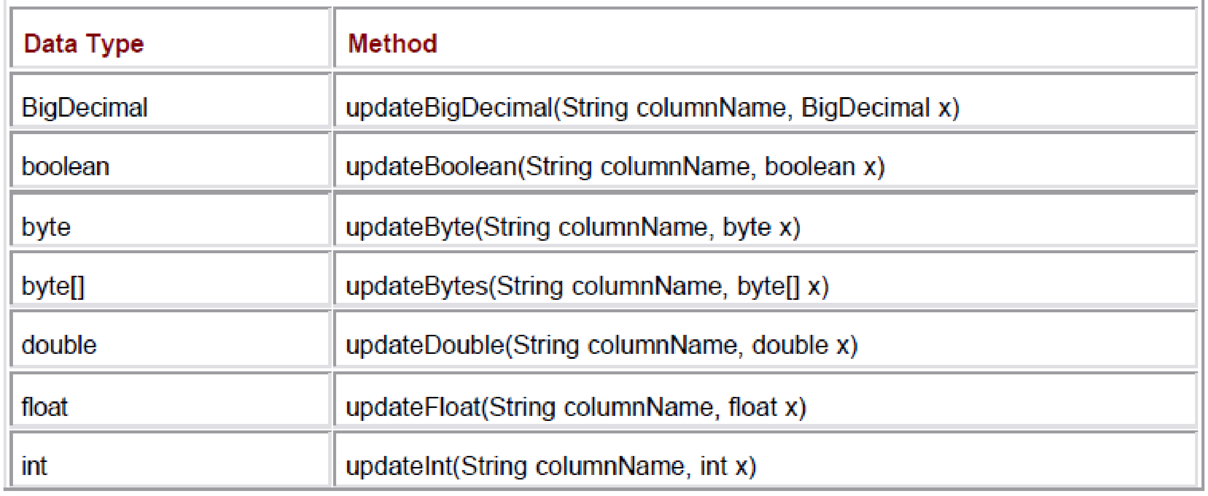
Since updates made to an Updatable ResultSet always affect the current row, you must make sure you have moved the cursor to the correct row prior to making an update.

Most of the ResultSet.update() methods take two parameters: the column to update and the new value to put in that column. As with the getter methods, the column may be specified using either the column name or the column number.

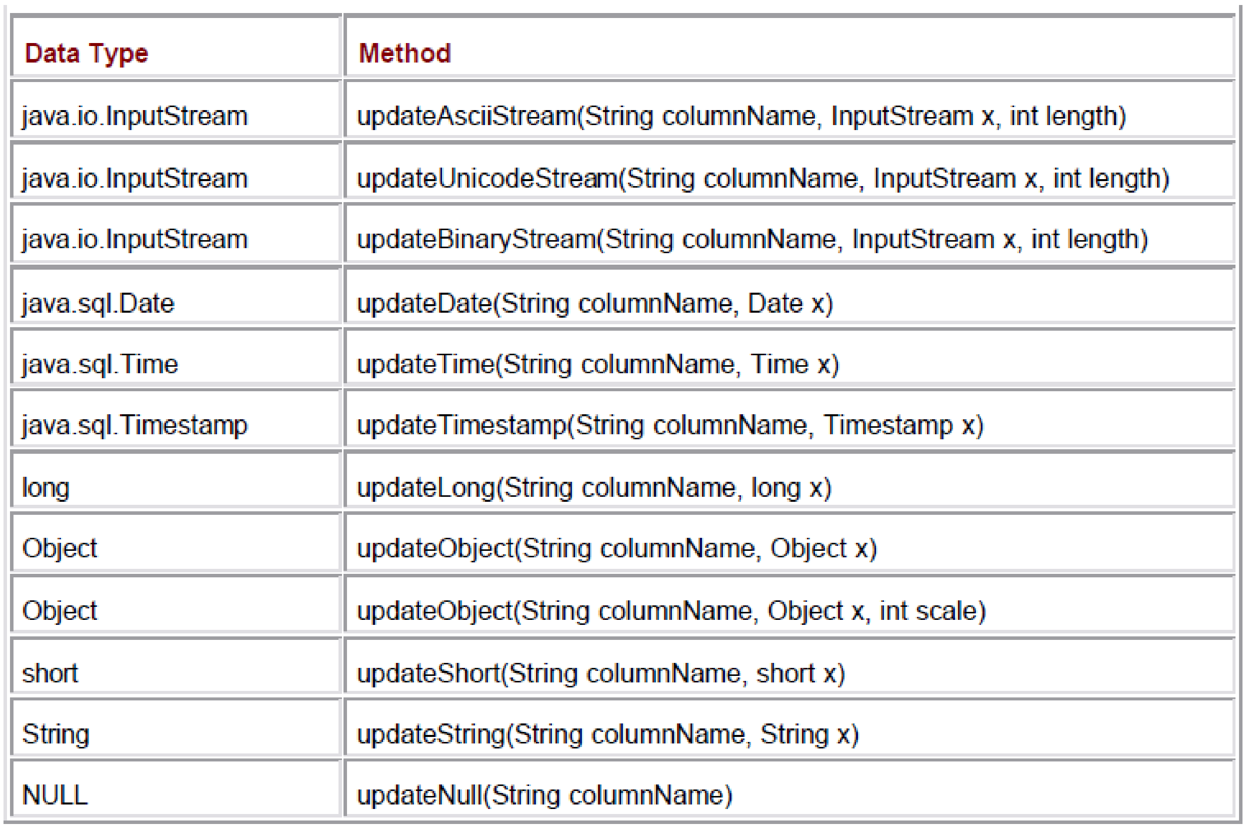
Table below summarizes the update methods for the UpdatableResultSet, showing only the variant using column name as the specifier for reasons of space.

to NULL.

Table 4-4: ResultSet Update Methods



Data Type	Method
BigDecimal	updateBigDecimal(String columnName, BigDecimal x)
boolean	updateBoolean(String columnName, boolean x)
byte	updateByte(String columnName, byte x)
byte[]	updateBytes(String columnName, byte[] x)
double	updateDouble(String columnName, double x)
float	updateFloat(String columnName, float x)
int	updateInt(String columnName, int x)



Data Type	Method
java.io.InputStream	updateAsciiStream(String columnName, InputStream x, int length)
java.io.InputStream	updateUnicodeStream(String columnName, InputStream x, int length)
java.io.InputStream	updateBinaryStream(String columnName, InputStream x, int length)
java.sql.Date	updateDate(String columnName, Date x)
java.sql.Time	updateTime(String columnName, Time x)
java.sql.Timestamp	updateTimestamp(String columnName, Timestamp x)
long	updateLong(String columnName, long x)
Object	updateObject(String columnName, Object x)
Object	updateObject(String columnName, Object x, int scale)
short	updateShort(String columnName, short x)
String	updateString(String columnName, String x)
NULL	updateNull(String columnName)

Note that after updating a column value in the ResultSet, you must call the ResultSet's updateRow() method to make a permanent change in the database before moving the cursor, since changes made using the update methods do not take effect until updateRow() is called.
If you move the cursor to another row before calling updateRow(), the updates will be lost, and the row will revert to its previous column values.
You can specifically cancel updates any time before calling updateRow() by calling the cancelRowUpdates() method. Once you have called updateRow(), however, the

cancelRowUpdates() method no longer works.

Inserting a New Row
In addition to supporting updates, an UpdatableResultSet supports the insertion and deletion of entire rows. The ResultSet object has a row called the insert row, which is, in effect, a dedicated row buffer in which you can build a new row.
The new row is created in a manner very similar to the row updates discussed earlier. Simply follow these steps:
Move the cursor to the insert row, which is done by calling the method moveToInsertRow().
2. Set a new value for each column in the row by using the appropriate update method.
3. Call the method insertRow() to insert the new row into the result set and, simultaneously, into the
database.

Eg:Consider the following program fragment
```
String query=”select * from stock”;
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection ("jdbc:odbc:Contacts");
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery(query);
rs.moveToInsertRow();
rs.updateInt("Contact_ID", 150);
rs.updateString("First_Name", "Nigel");
rs.updateString("Last_Name", "Thornebury");
rs.insertRow();
```

To move the cursor from the insert row back to the result set, you can use any of the methods that put the cursor on a specific row: first, last, beforeFirst, afterLast, and absolute. You can also use the methods previous and relative because the result set maintains a record of the current row while accessing the insert row.
In addition, you can use a special method: moveToCurrentRow(), which can be called only when the cursor is on the insert row. This method moves the cursor from the insert row back to the row that was previously the current row.

Deleting a Row
Deleting a row in an UpdatableResultSet is very simple. All you have to do is move the cursor to the row you want to delete and call the method deleteRow().
The example in the following code snippet shows how to delete the third row in a result set by getting the ResultSet object, moving the cursor to the third row, and using the deleteRow() method:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection ("jdbc:odbc:Contacts");
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery(query);
rs.absolute(3);
rs.deleteRow();
```

Be aware that different JDBC drivers handle deletions in different ways.
Some remove a deleted row so that it is no longer visible in a result set,
and others insert a blank row where the deleted row used to be.

Note: Remember that if you modify data in a ResultSet object, the change will always be visible if you close the ResultSet and reopen it by executing the same query again after the changes have been made.

Another way to get the most recent data is to use the method refreshRow(), which gets the latest values for a row straight from the database. This is done by positioning the cursor to the desired row and calling refreshRow(), as shown here:

```
rs.absolute(3);

rs.refreshRow();.
```

The various methods involved can be listed as follows:

• void moveToInsertRow()
moves the cursor to the insert row. The insert row is a
special row that is used for inserting new data with the updateXXX()
and insertRow() methods.

• void moveToCurrentRow()
moves the cursor back from the insert row to the row that it occupied when the moveToInsertRow () method was called.

• void insertRow()
inserts the contents of the insert row into the database and
the result set.

• void deleteRow()
deletes the current row from the database and the result
set.

• void updateXxx(int column, Xxx data)

• void updateXxx(String columnName, Xxx data)
(Xxx is a type such as int, double, String, Date, etc.)
update a field in the current row of the result set.

• void updateRow()
sends the current row updates to the database.

• void cancelRowUpdates()
cancels the current row updates.

rs.refreshRow()
refresh as per changes in database.

Java Transactions

If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions −

- To increase performance.
- To maintain the integrity of business processes.
- To use distributed transactions.

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit –

```
conn.setAutoCommit(false);
```

Commit & Rollback

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows –

```
conn.commit( );
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code –

```
conn.rollback( );
```

The following example illustrates the use of a commit and rollback object –

```
try{

   //Assume a valid connection object conn

   conn.setAutoCommit(false);

   Statement stmt = conn.createStatement();


   String SQL = "INSERT INTO Employees  " +

                "VALUES (106, 20, 'Rita', 'Tez')";

   stmt.executeUpdate(SQL);

   //Submit a malformed SQL statement that breaks

   String SQL = "INSERTED IN Employees  " +

                "VALUES (107, 22, 'Sita', 'Singh')";

   stmt.executeUpdate(SQL);

   // If there is no error.

   conn.commit();

}catch(SQLException se){

   // If there is any error.

   conn.rollback();
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

For a better understanding, let us study the Commit - Example Code.

Using Savepoints

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints −

- **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

The following example illustrates the use of a Savepoint object −

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   //set a Savepoint
   Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
   String SQL = "INSERT INTO Employees " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);

   //Submit a malformed SQL statement that breaks

   String SQL = "INSERTED IN Employees " +

                "VALUES (107, 22, 'Sita', 'Tez')";

   stmt.executeUpdate(SQL);

   // If there is no error, commit the changes.

   conn.commit();


}catch(SQLException se){

   // If there is any error.

   conn.rollback(savepoint1);

}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

JDBC - Batch Processing

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

- JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
- The **addBatch()** method of *Statement, PreparedStatement,* and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.

- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object −

- Create a Statement object using either *createStatement()* methods.
- Set auto-commit to false using *setAutoCommit()*.
- Add as many as SQL statements you like into batch using *addBatch()*method on created statement object.
- Execute all the SQL statements using *executeBatch()* method on created statement object.
- Finally, commit all the changes using *commit()* method.

Example

The following code snippet provides an example of a batch update using Statement object −

```
// Create statement object
Statement stmt = conn.createStatement();


// Set auto-commit to false
conn.setAutoCommit(false);


// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);


// Create one more SQL statement
```

```
String SQL = "INSERT INTO Employees (id, first, last, age) " +

             "VALUES(201,'Raj', 'Kumar', 35)";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);


// Create one more SQL statement

String SQL = "UPDATE Employees SET age = 35 " +

             "WHERE id = 100";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);


// Create an int[] to hold returned values

int[] count = stmt.executeBatch();


//Explicitly commit statements to apply changes

conn.commit();
```

.

Batching with PrepareStatement Object

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object −

1. Create SQL statements with placeholders.
2. Create PrepareStatement object using either *prepareStatement()*methods.
3. Set auto-commit to false using *setAutoCommit()*.
4. Add as many as SQL statements you like into batch using *addBatch()*method on created statement object.
5. Execute all the SQL statements using *executeBatch()* method on created statement object.
6. Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PrepareStatement object –

```
// Create SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) " +

             "VALUES(?, ?, ?, ?)";


// Create PrepareStatement object

PreparedStatemen pstmt = conn.prepareStatement(SQL);


//Set auto-commit to false

conn.setAutoCommit(false);


// Set the variables

pstmt.setInt( 1, 400 );

pstmt.setString( 2, "AAA" );

pstmt.setString( 3, "BBB" );

pstmt.setInt( 4, 33 );

// Add it to the batch

pstmt.addBatch();


// Set the variables

pstmt.setInt( 1, 401 );

pstmt.setString( 2, "XXX" );

pstmt.setString( 3, "YYYY" );

pstmt.setInt( 4, 31 );

// Add it to the batch

pstmt.addBatch();


//add more batches

.
```

```
.

.

.

//Create an int[] to hold returned values

int[] count = stmt.executeBatch();


//Explicitly commit statements to apply changes

conn.commit();
```

Using Advanced Data Types

The advanced data types introduced in this section give a relational database more flexibility in what can be used as a value for a table column. For example, a column can be used to store BLOB(binary large object) values, which can store very large amounts of data as raw bytes. A column can also be of type CLOB (character large object), which is capable of storing very large amounts of data in character format.

Mapping Advanced Data Types

The JDBC API provides default mappings for advanced data types specified by the SQL:2003 standard. The following list gives the data types and the interfaces or classes to which they are mapped:

- BLOB: Blob interface
- CLOB: Clob interface
- NCLOB: NClob interface
- ARRAY: Array interface
- XML: SQLXML interface
- Structured types: Struct interface
- REF(structured type): Ref interface
- ROWID: RowId interface
- DISTINCT: Type to which the base type is mapped. For example, a DISTINCT value based on a SQL NUMERIC type maps to a java.math.BigDecimal type because NUMERIC maps to BigDecimal in the Java programming language.
- DATALINK: java.net.URL object

Using Advanced Data Types

You retrieve, store, and update advanced data types the same way you handle other data types. You use either ResultSet.get*DataType* or CallableStatement.get*DataType* methods to retrieve them, PreparedStatement.set*DataType* methods to store them, and ResultSet.update*DataType* methods to update them. (The variable *DataType* is the name of a Java interface or class mapped to an advanced data type.) Probably 90 percent of the operations performed on advanced data types involve using the get*DataType*, set*DataType*, and update*DataType* methods. The following table shows which methods to use:

Advanced Data Type	get*DataType* **Method**	set*DataType* **method**	update*DataType* **Method**
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
NCLOB	getNClob	setNClob	updateNClob
ARRAY	getArray	setArray	updateArray
XML	getSQLXML	setSQLXML	updateSQLXML
Structured type	getObject	setObject	updateObject
REF(structured type)	getRef	setRef	updateRef
ROWID	getRowId	setRowId	updateRowId
DISTINCT	getBigDecimal	setBigDecimal	updateBigDecimal
DATALINK	getURL	setURL	updateURL

Note: The DISTINCT data type behaves differently from other advanced SQL data types. Being a user-defined type that is based on an already existing built-in types, it has no interface as its mapping in the Java programming language. Consequently, you use the method that corresponds to the Java type on which the DISTINCT data type is based. See Using DISTINCT Data Typefor more information.

For example, the following code fragment retrieves a SQL ARRAY value. For this example, suppose that the column SCORES in the table STUDENTS contains values of type ARRAY. The variable *stmt* is a Statement object.

```
ResultSet rs = stmt.executeQuery(
   "SELECT SCORES FROM STUDENTS " +
   "WHERE ID = 002238");
rs.next();
Array scores = rs.getArray("SCORES");
```

The variable *scores* is a logical pointer to the SQL ARRAY object stored in the table STUDENTS in the row for student 002238.

If you want to store a value in the database, you use the appropriate set method. For example, the following code fragment, in which *rs* is a ResultSet object, stores a Clob object:

```
Clob notes = rs.getClob("NOTES");
PreparedStatement pstmt =
   con.prepareStatement(
      "UPDATE MARKETS SET COMMENTS = ? " +
      "WHERE SALES < 1000000");
pstmt.setClob(1, notes);
pstmt.executeUpdate();
```

This code sets *notes* as the first parameter in the update statement being sent to the database. The Clob value designated by *notes* will be stored in the table MARKETS in column COMMENTSin every row where the value in the column SALES is less than one million.

Using Large Objects

An important feature of Blob, Clob, and NClob Java objects is that you can manipulate them without having to bring all of their data from the database server to your client computer. Some implementations represent an instance of these types with a locator (logical pointer) to the object in the database that the instance represents. Because a BLOB, CLOB, or NCLOB SQL object may be very large, the use of locators can make performance significantly faster. However, other implementations fully materialize large objects on the client computer.

If you want to bring the data of a BLOB, CLOB, or NCLOB SQL value to the client computer, use methods in the Blob, Clob, and NClob Java interfaces that are provided for this purpose. These large object type objects materialize the data of the objects they represent as a stream.

Adding Large Object Type Object to Database

The Clob Java object myClob contains the contents of the file specified by fileName.

```
public void addRowToCoffeeDescriptions(
    String coffeeName, String fileName)
    throws SQLException {

    PreparedStatement pstmt = null;
    try {
        Clob myClob = this.con.createClob();
        Writer clobWriter = myClob.setCharacterStream(1);
        String str = this.readFile(fileName, clobWriter);
        System.out.println("Wrote the following: " +
            clobWriter.toString());

        if (this.settings.dbms.equals("mysql")) {
            System.out.println(
                "MySQL, setting String in Clob " +
                "object with setString method");
            myClob.setString(1, str);
        }
        System.out.println("Length of Clob: " + myClob.length());

        String sql = "INSERT INTO COFFEE_DESCRIPTIONS " +
                     "VALUES(?,?)";

        pstmt = this.con.prepareStatement(sql);
        pstmt.setString(1, coffeeName);
        pstmt.setClob(2, myClob);
        pstmt.executeUpdate();
    } catch (SQLException sqlex) {
        JDBCTutorialUtilities.printSQLException(sqlex);
    } catch (Exception ex) {
      System.out.println("Unexpected exception: " + ex.toString());
    } finally {
        if (pstmt != null)pstmt.close();
    }
}
```

The following line creates a Clob Java object:

```
Clob myClob = this.con.createClob();
```

The following line retrieves a stream (in this case a Writer object named clobWriter) that is used to write a stream of characters to the Clob Java object myClob. The method ClobSample.readFile writes this stream of characters; the stream is from the file specified by the String fileName. The method argument 1 indicates that the Writer object will start writing the stream of characters at the beginning of the Clob value:

```
Writer clobWriter = myClob.setCharacterStream(1);
```

The ClobSample.readFile method reads the file line-by-line specified by the file fileName and writes it to the Writer object specified by writerArg:

```
private String readFile(String fileName, Writer writerArg)
      throws FileNotFoundException, IOException {

   BufferedReader br = new BufferedReader(new FileReader(fileName));
   String nextLine = "";
   StringBuffer sb = new StringBuffer();
   while ((nextLine = br.readLine()) != null) {
      System.out.println("Writing: " + nextLine);
      writerArg.write(nextLine);
      sb.append(nextLine);
   }
   // Convert the content into to a string
   String clobData = sb.toString();

   // Return the data.
   return clobData;
}
```

The following excerpt creates a PreparedStatement object pstmt that inserts the Clob Java object myClob into COFFEE_DESCRIPTIONS:

```
PreparedStatement pstmt = null;
// ...
String sql = "INSERT INTO COFFEE_DESCRIPTIONS VALUES(?,?)";
pstmt = this.con.prepareStatement(sql);
pstmt.setString(1, coffeeName);
pstmt.setClob(2, myClob);
pstmt.executeUpdate();
```

Retrieving CLOB Values

The method ClobSample.retrieveExcerpt retrieves the CLOB SQL value stored in the COF_DESC column of COFFEE_DESCRIPTIONS from the row whose column value COF_NAME is equal to the String value specified by the coffeeName parameter:

```
public String retrieveExcerpt(String coffeeName, int numChar)
    throws SQLException {

    String description = null;
    Clob myClob = null;
    PreparedStatement pstmt = null;

    try {
        String sql =
            "select COF_DESC " +
            "from COFFEE_DESCRIPTIONS " +
            "where COF_NAME = ?";

        pstmt = this.con.prepareStatement(sql);
        pstmt.setString(1, coffeeName);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            myClob = rs.getClob(1);
            System.out.println("Length of retrieved Clob: " +
                myClob.length());
        }
        description = myClob.getSubString(1, numChar);
    } catch (SQLException sqlex) {
        JDBCTutorialUtilities.printSQLException(sqlex);
    } catch (Exception ex) {
        System.out.println("Unexpected exception: " + ex.toString());
    } finally {
        if (pstmt != null) pstmt.close();
    }
    return description;
}
```

The following line retrieves the Clob Java value from
the ResultSet object rs:

```
myClob = rs.getClob(1);
```

The following line retrieves a substring from the myClob object. The substring begins at the first character of the value of myClob and has up to the number of consecutive characters specified in numChar, where numChar is an integer.

```
description = myClob.getSubString(1, numChar);
```

Adding and Retrieving BLOB Objects

Adding and retrieving BLOB SQL objects is similar to adding and retrieving CLOB SQL objects. Use the Blob.setBinaryStream method to retrieve an OutputStream object to write the BLOBSQL value that the Blob Java object (which called the method) represents.

Releasing Resources Held by Large Objects

Blob, Clob, and NClob Java objects remain valid for at least the duration of the transaction in which they are created. This could potentially result in an application running out of resources during a long running transaction. Applications may release Blob, Clob, and NClob resources by invoking their free method.

In the following excerpt, the method Clob.free is called to release the resources held for a previously created Clob object:

```
Clob aClob = con.createClob();
int numWritten = aClob.setString(1, val);
aClob.free();
```