CHAPTER - 23

JAVA DATABASE CONNECTIVITY (JDBC)

KEY OBJECTIVES:_____

After completing this chapter readers will be able to—

- get an idea about different types of JDBC drivers
- learn how to use JDBC technology in Java Server Pages
- Know how to load JDBC drivers
- retrieve database metadata
- call stored procedures
- understand scrollable and updatable result set

23.1 Introduction

Many Java applications need access to databases. **J**ava **D**ata**B**ase **C**onnectivity (JDBC) allows us to access databases through Java programs. It provides Java classes and interfaces to fire SQL and PL/SQL statements, process results (if any), and perform other operations common to databases. Since Java Server Pages can contain Java code embedded in them, it is also possible to access databases from Java Server Pages. The classes and interfaces for database connectivity are provided as a separate package, `java.sql`.

23.2 JDBC Drivers

A Java application can access almost all types of databases such as relational, object, and object-relational. The access to a specific database is accomplished using a set of Java interfaces, each of which is implemented by different vendors differently. A Java class that provides interfaces to a specific database is called JDBC driver. Each database has its own set of JDBC drivers. Users need not bother about the implementation of those Java classes. They can concentrate on developing database applications. Those drivers are provided (generally freely) by database vendors. This way, JDBC hides the underlying database architecture. JDBC drivers provided by database vendors convert database access requests to database-specific APIs.

JDBC drivers are classified into four categories depending upon the way they work.

23.2.1 JDBC-ODBC bridge (Type 1)

This is the *Type 1* driver. This type of drivers cannot talk to the database directly. It needs an intermediate ODBC (**O**pen **D**ata**B**ase **C**onnectivity) driver, with which it forms a kind of bridge. The driver translates JDBC function calls to ODBC method calls. ODBC makes use of native libraries of the operating system and is hence platform-dependent. For this mechanism to function correctly, the ODBC driver must be available in the client machine and must also be configured correctly, which is generally a long and tedious process. For this reason, the Type 1 driver is used for experimental purposes or when no other JDBC driver is available. Sun provides a Type 1 JDBC driver with JDK 1.1 or later.

23.2.2 Native-API, Partly Java (Type 2)

This is very similar to the Type 1 driver. However, it does not forward the JDBC call to the ODBC driver. Instead, it translates JDBC calls to database-specific native API calls. This driver is not a pure Java driver, as it interfaces with non-Java APIs that communicates with the database. This approach is a little bit faster than the earlier one, as it interfaces directly with the database through the native APIs. However, it has limitations similar to the previous one. This means that the client must have vendor-specific native APIs installed and configured in it.

23.2.3 Middleware, Pure Java (Type 3)

In this case, the JDBC driver forwards the JDBC calls to some middleware server [Figure 23.1: (ii)] using a database-independent network protocol. The middleware server acts as a *gateway* for multiple (possibly different) database servers and can use different database-specific protocols to connect to different database servers. This intermediate server sends each client request to a specific database. The results are then sent back to the intermediate server, which in turn sends the result back to the client. This approach hides the details of connections to the database servers and makes it possible to change the database servers without affecting the client.

23.2.4 Pure Java Driver (Type 4)

These types of drivers communicate with the database directly by making socket connections. It has distinct advantages over other mechanisms, in terms of performance and development time. Since, it talks with the database server directly, no other subsidiary driver is needed. In this book, we shall use only the Type 4 driver.

23.3 JDBC Architecture

The JDBC architecture is sometimes classified as: two-tier and three-tier. Type 2 and 4 drivers use two-tier and Type 1 and 3 use three-tier architecture. Figure 23.1: (i) and (ii) show the JDBC two-tier and three-tier architectures.

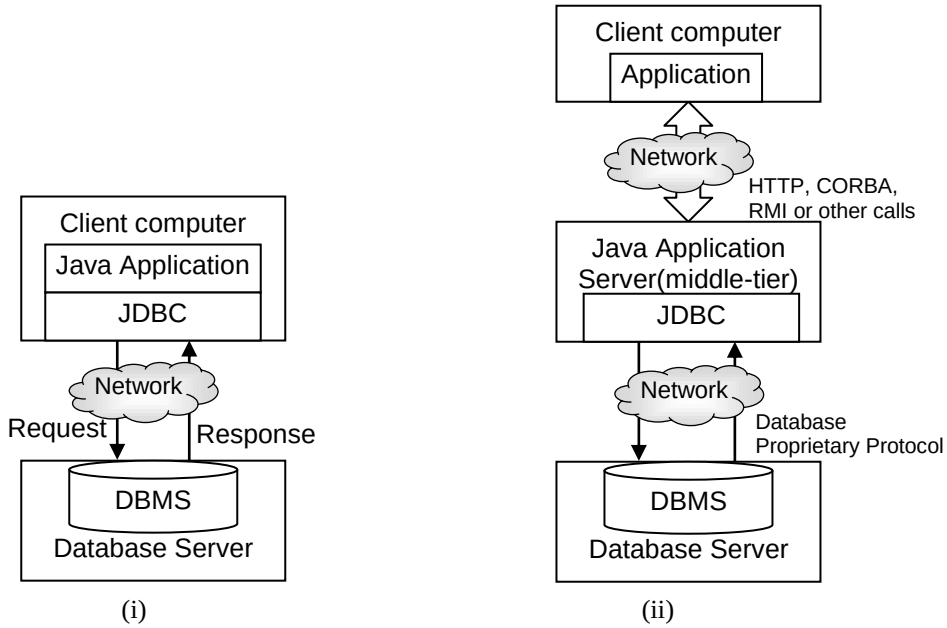

Figure 23.1: JDBC architecture (i) two tier (ii) three-tier

23.14 JDBC Classes and Interfaces

Java provides an API for accessing and processing data stored in a data source (usually a relational database). A summary of JDBC classes and interfaces with a brief description is shown in Table 23.1:

Table 23.1: Java JDBC classes and interfaces

Class/Interface	Description
DriverManager	The basic service for managing a set of JDBC drivers
Connection	A connection (session) with a specific database
Statement	The object used for executing a static SQL statement and returning the results it produces
ResultSet	A table of data representing a database result set, which is usually generated by executing a statement that queries the database
PreparedStatement	An object that represents a precompiled SQL statement
CallableStatement	The interface used to execute SQL stored procedures.
DatabaseMetaData	Comprehensive information about the database as a whole
ResultSetMetaData	An object that can be used to get information about the types and properties of the columns in a ResultSet object.

In this chapter, we shall demonstrate how to use JDBC with respect to JSP.

23.15 Basic Steps

The following basic steps are followed to work with JDBC:

- Loading a Driver
- Making a connection
- Executing an SQL statement

23.16 Loading a Driver

You have to first download an appropriate driver depending upon the database you want to connect. Sun provides a Type 1 driver bundled with the JDK 1.1 or later. Other types of drivers are database-specific and must be downloaded.

The latest version, Type 4 MySQL JDBC driver, can be downloaded from the following site:

```
http://dev.mysql.com/downloads/connector/j/#downloads
```

Download the `.zip` or `.tar.gz` file containing the jar (**j**ava **ar**chive) file `mysql-connector-java-5.1.26-bin.jar`.

The latest version Type 4 JDBC driver for Oracle can be downloaded from the following site:

```
http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html
```

Download the appropriate driver depending upon the JDK and Oracle version you are using. For example, the binary driver file for Oracle 12c and JDK 1.7 or later is `ojdbc7.jar`.

Once you have downloaded the appropriate .jar file, put it in Tomcat's `lib` directory and restart the web server.

If you are developing simple Java database applications, put this .jar file in the `CLASSPATH` environment variable.

So far, we have downloaded and installed the JDBC driver. For it to start functioning, an instance of the driver has to be created and registered with the `DriverManager` class so that it can translate the JDBC call to the appropriate database call. The JDBC class `DriverManager` is an important class in the `java.sql` package. It interfaces between the Java application and the JDBC driver. This class manages the set of JDBC drivers installed on the system. It has many other useful methods, some of which will be discussed in Section 26.17.

One way to register a driver with the driver manager is to use the static with a driver class name as an argument. For example, the Type 1 driver provided by sun can be instantiated and registered with the driver manager as follows:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

The method `forName()` creates an instance of the class whose name is specified as an argument using its default constructor. The instance created in this fashion must register itself with the `DriverManager` class. The .jar file for MySQL contains two driver class files with the name `Driver.class`, one in the `com.mysql.jdbc` package and the other in the `org.gjt.mm.mysql` package. So, you may use any one of the following:

```
Class.forName("com.mysql.jdbc.Driver");
Class.forName("org.gjt.mm.mysql.Driver");
```

One can perform this registration procedure by explicitly creating an instance and passing it to the static `registerDriver()` method of the `DriverManager` class. The method `registerDriver()`

in turn registers the driver with the driver manager. Some JDBC vendors such as Oracle recommend the latter mechanism.

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

A similar procedure can be followed for other drivers as well. The implementation of the MySQL driver file `com.mysql.jdbc.Driver` looks like this.

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
   static {
      try {
         java.sql.DriverManager.registerDriver(new Driver());
      } catch (SQLException E) {
         throw new RuntimeException("Can't register driver!");
      }
   }
   public Driver() throws SQLException {}
}
```

Since the static block registers the driver with the driver manager automatically, only creating an instance is sufficient. So, one might use the following code as well:

```
new com.mysql.jdbc.Driver();
```

Now, the driver is ready to translate the JDBC call.

23.17 Making a connection

Once a driver is instantiated and registered with the driver manager, the connection to the database can be established using methods provided by the `DriverManager` class. For each connection created, `DriverManager` makes use of the appropriate driver registered to it. The following methods are available on the `DriverManager` class to establish a connection. All methods return a `Connection` object on successful creation of the connection.

```
public static Connection getConnection(String url, String login, String passwd)
public static Connection getConnection(String url)
public static Connection getConnection(String url, Properties)
```

The `Connection` object encapsulates the session/connection to a specific database. It is used to fire SQL statements as well as commit or roll back database transactions. It also allows us to collect useful information about the database dynamically and to write custom applications. Many connections can be established to a single database server or different database servers.

The primary argument that the `getConnection()` method takes is a database URL. This argument identifies a database uniquely. `DriverManager` uses this URL to find a suitable JDBC driver installed earlier, which recognizes the URL and uses this driver to connect to the corresponding database.

The URL always starts with `jdbc:`. The format of the rest of the JDBC URL varies widely for different databases. Some are mentioned in Table 23.2: The format of the MySQL JDBC URL is as follows:

```
jdbc:mysql://[host]:[port]/[database]
```

Here, `host` is the name (or IP address) of the machine running the database at the port number `port` and `database` is a name of a database. Suppose a MySQL database, `test`, is running in a machine, `uroy`, at port `3306`, the corresponding URL will be

```
jdbc:mysql://uroy:3306/test
```

A database connection can be established using this URL as follows:

```
Connection con = DriverManager.getConnection("jdbc:mysql://uroy:3306/test",
"root", "nbuser");
```

Similarly, the following code segment creates a database connection to the Oracle database `mirora` running in the machine `miroracle` at port `1521`.

```
Connection  con =
DriverManager.getConnection("jdbc:oracle:thin:@miroracle:1521:mirora", "scott",
"tiger");
```

Table 21.5 shows the JDBC URL formats.

Table 23.2: JDBC URL format

MySQL		
Jar file	mysql-connector-java-nn-bin.jar	
Download URL	http://dev.mysql.com/downloads/connector/j/5.1.html	
Driver	com.mysql.jdbc.Driver	
URL format	jdbc:mysql://[host]:[port]/[database]	
Sample URL	jdbc:mysql://uroy:3306/test	
	jdbc:mysql://localhost:3306/sample	
Oracle		
Jar file	ojdbc6.jar (Java 1.6) ojdbc7.jar (Java 1.7)	
Download URL	http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html	
Driver	oracle.jdbc.driver.OracleDriver	
URL format	jdbc:oracle:[type]:@[host]:[port]:[service] jdbc:oracle:[type]:[host]:[port]:[SID] jdbc:oracle:[type]:[TNSName]	
Sample URL	thin	jdbc:oracle:thin:@miroracle:1521: ORCL_SVC jdbc:oracle:thin:@172.16.4.243:1521: ORCL_SID jdbc:oracle:thin:@(description=(address=(host=localhost)(protocol=tcp)(port=1521))(connect_data=(sid=ORCL))) jdbc:oracle:thin:@TNS-NAME
	Oci	jdbc:oracle:oci:@miroracle:1521: ORCL_SVC jdbc:oracle:oci:@172.16.4.243:1521: ORCL_SID jdbc:oracle:oci:@(description=(address=(host=localhost)(protocol=tcp)(port=1521))(connect_data=(sid=ORCL))) jdbc:oracle:oci:@TNS-NAME
Sun JDBC-ODBC Bridge		
Jar file	Bundled with JDK	
Driver	Sun.jdbc.odbc.JdbcOdbcDriver	
URL format	JDBC:ODBC:[data source name]	
Sample URL	JDBC:ODBC:test	
DB2		
Jar file	db2jcc.jar	

Download URL	http://www-01.ibm.com/software/data/db2/ad/java.html
Driver	Com.ibm.db2.jdbc.net.DB2Driver
URL format	Jdbc:db2://[host]:[port]/[database]
Sample URL	jdbc:db2://172.16.4.243:50000/test
Pervasive	
Jar file	pvjdbc2.jar
Driver	com.pervasive.jdbc.v2.Driver
Download URL	http://www.pervasive.com/developerzone/access_methods/jdbc.asp
URL format	jdbc:pervasive://[host]:[port]/[database]
Sample URL	jdbc:pervasive://uroy:1583/sample
PostgreSQL	
Jar file	postgresql-nn.jdbc3.jar
Download URL	http://jdbc.postgresql.org/download.html
Driver	org.postgresql.Driver
URL format	jdbc:postgresql://[host]:[port]/[database]
Sample URL	jdbc:postgresql://[localhost]:[5432]/[test]
JavaDB/Derby	
Jar file	derbyclient.jar
Download URL	http://db.apache.org/derby/derby_downloads.html
Driver	org.apache.derby.jdbc.ClientDriver
URL format	jdbc:derby:net://[host]:[port]/[database]
Sample URL	jdbc:derby:net://[172.16.4.243]:[1527]/[sample]

The second overloaded version of the `getConnection()` method takes only a string argument. This argument must contain URL information, together with other parameters such as user name and password. The parameters are passed as a name–value pair separated by "&" using the same syntax as the HTTP URL. The general syntax of such a URL is as follows:

```
basicURL?param1=value1&param2=value2…
```

Following is an example of such a string argument for the MySQL database.

```
jdbc:mysql://uroy:3306/test?user=root&password=nbuser
```

Alternatively, parameters can be put in a `java.util.Properties` object and the object can be passed to the `getConnection()` method. Following is an example using Properties.

```
String url = "jdbc:mysql://uroy:3306/test";
java.util.Properties p = new java.util.Properties();
p.setProperty("user", "root");
p.setProperty("password", "nbuser");
Connection con = DriverManager.getConnection(url, p);
```

23.18 Execute SQL statement

Once a connection to the database is established, we can interact with the database. The `Connection` interface provides methods for obtaining different statement objects that are used to fire SQL statements via the established connection. The `Connection` object can be used for other

purposes such as gathering database information, and committing or rolling back a transaction. The following section describes different types of statement objects and their functionality.

23.19 SQL Statements

The `Connection` interface defines the following methods to obtain statement objects.

```
Statement createStatement()
Statement createStatement(int resultSetType,
                          int resultSetConcurrency)
Statement createStatement(int resultSetType,
                          int resultSetConcurrency,
                          int resultSetHoldability)

PreparedStatement prepareStatement(java.lang.String)
PreparedStatement prepareStatement(String sql,
                                   int resultSetType,
                                   int resultSetConcurrency)
PreparedStatement prepareStatement(String sql,
                                   int resultSetType,
                                   int resultSetConcurrency,
                                   int resultSetHoldability)

CallableStatement prepareCall(java.lang.String)
CallableStatement prepareCall(String sql,
                              int resultSetType,
                              int resultSetConcurrency)
CallableStatement prepareCall(String sql,
                              int resultSetType,
                              int resultSetConcurrency,
                              int resultSetHoldability)
```

The JDBC `Statement`, `CallableStatement`, and `PreparedStatement` interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

23.19.1Simple Statement

The `Statement` interface is used to execute static SQL statements A `Statement` object is instantiated using the `createStatement()` method on the `Connection` object as follows:

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

This `Statement` object defines the following methods to fire different types of SQL commands on the database.

`executeUpdate()`

This method is used to execute DDL (CREATE, ALTER, and DROP), DML (INSERT, DELETE, UPDATE, etc.) and DCL statements. In general, if an SQL command changes the database, the `executeUpdate()` method is used. The return value of this method is the number of rows affected.

Assume that `stmt` is a `Statement` object. The following code segment first creates a table named `accounts`.

```
String create =   "CREATE TABLE accounts (               "+
                  "  accNum      integer   primary key,  "+
                  "  holderName  varchar(20),            "+
                  "  balance     integer                 "+
                  ")";
```

```
stmt.executeUpdate(create);
```

Once the table is created, data can be inserted into it using the following code segment.

```
String insert = "INSERT INTO accounts VALUES(1,'Uttam K. Roy', 10000)";
stmt.executeUpdate(insert);
insert = "INSERT INTO accounts VALUES(2,'Bibhas Ch. Dhara', 20000)";
stmt.executeUpdate(insert);
```

The following JSP page (`DDL.jsp`) creates a table accounts and insert two rows in the table. Make sure that the MySQL JDBC driver is in the `/lib` directory under the Tomcat installation directory.

```
<!--DDL.jsp-->
<%@page import="java.sql.*"%>
<%
  try {
    new com.mysql.jdbc.Driver();
    String url = "jdbc:mysql://uroy:3306/test?user=root&password=nbuser";
    Connection conn = DriverManager.getConnection(url);
    Statement stmt = conn.createStatement();

    String create = "CREATE TABLE accounts (          " +
          "  accNum     integer   primary key,  " +
          "  holderName varchar(20),         " +
          "  balance    integer        " +
          ")";
    stmt.executeUpdate(create);

    String insert = "INSERT INTO accounts VALUES(1,'Uttam K. Roy', 10000)";
    stmt.executeUpdate(insert);
    insert = "INSERT INTO accounts VALUES(2,'Bibhas Ch. Dhara', 20000)";
    stmt.executeUpdate(insert);
    stmt.close();
    conn.close();
    out.println("Created table accounts");
  } catch (Exception e) {
    out.println(e);
  }
%>
```

`executeQuery()`

This is used for DQL statements such as SELECT. Remember, DQL statements only read data from database tables; it cannot change database tables. So, the return value of this method is a set of rows that is represented as a `ResultSet` object.

The result of the `executeQuery` method is stored in an object of type `ResultSet`. This result set object looks very much similar to a table and hence has a number of rows. A particular row is selected by setting a cursor associated with this result set. A cursor is something like a pointer to the rows. Once the cursor is set to a particular row, individual columns are retrieved using the methods provided by the `ResultSet` interface. The cursor is placed before the first row of result set when it is created first. JDBC 1.0 allows us to move the cursor only in the forward direction using the method `next()`. JDBC 2.0 allows us to move the cursor both forward and backward as well as to move to a specified row relative to the current row. These types of result sets are called scrollable result sets, which will be discussed in Section 21.22.

Since the cursor does not point to any row (it points to a position before the first row), users have the responsibility to set the cursor to a valid row to retrieve data from it. To retrieve data from a column, methods of the form `getX()` are used, where `X` is the data type of the column. Following is an example that retrieves information from the `accounts` table created earlier.

```
String query = "SELECT * FROM accounts";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
   out.println(rs.getString("accNum"));
   out.println(rs.getString("holderName"));
   out.println(rs.getString("balance"));
}
```

`execute()`

Sometimes, users want to execute SQL statements whose type (DDL, DML, DCL, or DQL) is not known in advance. This may happen particularly when statements are obtained from another program. In that case, users cannot decide which method they should use. In such cases, the `execute()` method is used. It can be used to execute any SQL commands. Since it allows us to execute any SQL commands, the result can either be a `ResultSet` object or an integer. However, how does a user know it? Fortunately, this method returns a Boolean value, which indicates the return type. The return value `true` indicates that the result is a `ResultSet` object, which can be obtained by calling its `getResultSet()` method. On the other hand, if the return value is `false`, the result is an update count, which can be obtained by calling the `getUpdateCount()` method.

The following JSP page (`execute.jsp`) takes an arbitrary SQL statement as a parameter and fires this SQL statement on the database.

```
<!--execute.jsp-->
<%@page import="java.sql.*"%>
<%
   response.setHeader("Pragma", "no-cache");
   response.setHeader("Cache-Control", "no-cache");
   response.setDateHeader("Expires", -1);
   try {
      String query = request.getParameter("sql");
      if (query != null) {
         new com.mysql.jdbc.Driver();
         String url = "jdbc:mysql://uroy:3306/test";
         Connection con = DriverManager.getConnection(url, "root", "nbuser");
         Statement stmt = con.createStatement();
         if (stmt.execute(query) == false) {
            out.println(stmt.getUpdateCount() + " rows affected");
         }
         else {
            ResultSet rs = stmt.getResultSet();
            ResultSetMetaData md = rs.getMetaData();
            out.println("<table border=\"1\"><tr>");
            for (int i = 1; i <= md.getColumnCount(); i++) {
               out.print("<th>" + md.getColumnName(i) + "</th>");
            }
            out.println("</tr>");
            while (rs.next()) {
               out.println("<tr>");
               for (int i = 1; i <= md.getColumnCount(); i++) {
                  out.print("<td>" + rs.getString(i) + "</td>");
               }
               out.println("</tr>");
            }
            out.println("</table>");
            rs.close();
         }
         stmt.close();
         con.close();
      }
   }
   catch (Exception e) {
```

```
      out.println(e);
   }
%>
<form name="sqlForm" method="post">
   SQL statement:<br><input type="text" name="sql" size="50"><br />
   <input type="reset"><input type="submit" value="Execute">
</form>
```

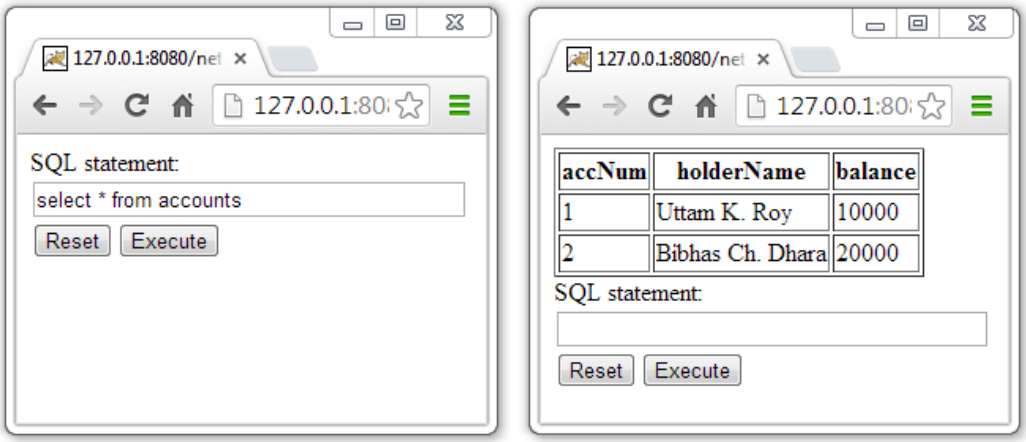

Figure 23.2: Executing SQL statements

Note that this JSP page allows you to fire any valid SQL query, including DML. So, the JSP page must perform user verification before providing this interface.

23.19.2Atomic transaction

The database transaction made by the `executeUpdate()` method is committed automatically. This may lead to data inconsistency if a series of related statements are executed. Consider the following simple table for a banking application.

```
accounts(accNum, holderName, balance)
```

The bank manager wants to write a java program to transfer some amount of money `amount` from the source account `src` to the destination account `dst`. The basic task of this program will be to subtract `amount` from the source account balance and add `amount` to the destination account balance. A sample JSP page looks like this:

```
<!--manager.jsp-->
<%@page import="java.sql.*"%>
<%!
   Connection con;
   Statement stmt;
   String query;
   public void jspInit() {
      try {
         new com.mysql.jdbc.Driver();
         String url = "jdbc:mysql://uroy:3306/test";
         con = DriverManager.getConnection(url, "root", "nbuser");
         stmt = con.createStatement();
      }catch(Exception e) {}
   }
   public boolean transfer(int src, int dst, int amount) {
```

```
      try {
         query = "SELECT balance FROM accounts WHERE accNum=" + src;
         ResultSet rs = stmt.executeQuery(query);
         rs.next();
         int srcBal = rs.getInt("balance") - amount;
         query = "SELECT balance FROM accounts WHERE accNum=" + dst;
         rs = stmt.executeQuery(query);
         rs.next();
         int dstBal = rs.getInt("balance") + amount;
         return doTransfer(src, dst, srcBal, dstBal);
      } catch (SQLException e) {
         return false;
      }
   }
   public boolean doTransfer(int src, int dst, int srcBal, int dstBal) {
   try {
      query = "UPDATE accounts SET balance=" + srcBal + " WHERE accNum=" + src;
      stmt.executeUpdate(query);
      query = "UPDATE accounts SET balance=" + dstBal + " WHERE accNum=" + dst;
      //If anything goes wrong here, destination account will contain wrong
      //result.
      stmt.executeUpdate(query);
      return true;
   } catch (SQLException e) {
      return false;
   }
   }
%>
<%
try {
   int src = Integer.parseInt(request.getParameter("src"));
   int dst = Integer.parseInt(request.getParameter("dst"));
   int amount = Integer.parseInt(request.getParameter("amount"));
   transfer(src, dst, amount);
}catch(Exception e) {out.println(e);}
%>
```

Note that source and destination accounts must be updated atomically. However, if anything goes wrong after updating the source account and before updating the destination account in the `doTransfer()` method, the destination account will hold an incorrect balance.

This problem can be solved using the `autoCommit()` method available on the `Connection` object. First the `autoCommit` flag of the `Connection` object is set to false. At the end of execution of all related statements, the transaction is committed. If anything goes wrong during the execution of those statements, it can be caught and the transaction gets rolled back accordingly. This way a set of related operations can be performed atomically.

The correct `doTransfer()` method looks like this:

```
public boolean doTransfer(int src, int dst, int srcBal, int dstBal) {
   try {
      con.setAutoCommit(false);
      query = "UPDATE accounts SET balance=" + srcBal + " WHERE accNum=" + src;
      stmt.executeUpdate(query);
      query = "UPDATE accounts SET balance=" + dstBal + " WHERE accNum=" + dst;
      stmt.executeUpdate(query);
      con.commit();
      return true;
   } catch (SQLException e) {
      try {
         con.rollback();
      } catch (SQLException e1) {
```

```
        }
        return false;
      }
    }
```

`executeBatch()`

This method allows us to execute a set of related commands as a whole. Commands to be fired on the database are added to the `Statement` object one by one using the method `addBatch()`. It is always safe to clear the `Statement` object using the method `clearBatch()` before adding any command to it. Once all commands are added, `executeBatch()` is called to send them as a unit to the database. The DBMS executes the commands in the order in which they were added. Finally, if all commands are successful, it returns an array of update counts. To allow correct error handling, we should always set auto-commit mode to `false` before beginning a batch command.

Following is the method `doTransfer`, rewritten using this mechanism.

```
public boolean doBatchTransfer(int src, int dst, int srcBal, int dstBal) {
  try {
    String query;
    con.setAutoCommit(false);
    stmt.clearBatch();
    query = "UPDATE accounts SET balance=" + srcBal + " WHERE accNum=" + src;
    stmt.addBatch(query);
    query = "UPDATE accounts SET balance=" + dstBal + " WHERE accNum=" + dst;
    stmt.addBatch(query);
    stmt.executeBatch();
    con.commit();
    return true;
  } catch (SQLException e) {
    try {
      con.rollback();
    } catch (SQLException e1) {
    }
    return false;
  }
}
```

Since all the commands are sent as a unit to the database for execution, it improves the performance significantly.

23.19.3Pre-compiled Statement

When an SQL statement is fired to the database for execution using the `Statement` object the following steps get executed:

- DBMS checks the syntax of the statement being submitted.
- If the syntax is correct, it executes the statement.

DBMS compiles every statement unnecessarily, even if users want to execute the *same* SQL statement repeatedly with different data items. This creates significant overhead, which can be avoided using the `PreparedStatement` object.

A `PreparedStatement` object is created using the `prepareStatement()` method of the `Connection` object. An SQL statement with placeholders (?) is supplied to the method `Connection.prepareStatement()` when a `PreparedStatement` object is created. This SQL statement, together with the placeholders is sent to the DBMS. DMBS, in turn, compiles the statement and if everything is correct, a `PreparedStatement` object is created. This means that a `PreparedStatement` object contains an SQL statement whose syntax has already been checked

and hence is called pre-compiled statement. This SQL statement is then fired repeatedly, with placeholders substituted by different data items. Note that `PreparedStatement` is useful only if the same SQL statement is executed repeatedly with different parameters. Otherwise, it behaves exactly like `Statement` and no benefit can be obtained.

The following example creates a `PreparedStatement` object:

```
PreparedStatement ps = con.prepareStatement("INSERT INTO user values(?,?)");
```

The SQL statement has two placeholders, whose values will be supplied whenever this statement is sent for execution. Placeholders are substituted using methods of the form `setX()`, where `X` is the data type of the value used to substitute. These methods take two parameters. The first parameter indicates the index of the placeholder to be substituted and the second one indicates the value to be used for substitution. The following example substitutes the first placeholder with "user1".

```
ps.setString(1, "user1");
```

Consider a file, `question.txt`, which contains questions of the form `question_no:question_string` as follows:

```
1:What is the full form of JDBC?
2:How is a PreparedStatement created?
…
```

The following example inserts questions and their numbers stored in this file in the table `questions`.

```
PreparedStatement ps = con.prepareStatement("INSERT INTO questions
values(?,?)");
BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream("question.txt")));
String line = br.readLine();
while (line != null) {
   StringTokenizer st = new StringTokenizer(line, ":");
   String qno = st.nextToken();
   String question = st.nextToken();
   ps.setString(1, qno);
   ps.setString(2, question);
   ps.executeUpdate();
   line = br.readLine();
}
```

`PreparedStatement` has another important role in executing parameterized SQL statements. Consider this solution using the `Statement` object.

```
Statement stmt = con.createStatement();
BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(application.getRealPath("/")+"question.txt")));
String line = br.readLine();
while (line != null) {
   StringTokenizer st = new StringTokenizer(line, ":");
   String qno = st.nextToken();
   String question = st.nextToken();
   String query = "INSERT INTO questions values("+qno+",'"+question+"')";
   stmt.executeUpdate(query);
   line = br.readLine();
}
```

This code segment will work fine, provided the question does not contain characters such as "'". For example, if the file `question.txt` contains a line "3:What's JDBC?", the value of the query will be

```
INSERT INTO questions values(3,'What's JDBC?')
```

This is an invalid query due the "'" character in the word "What's". If you try, you will get an error message like this:

```
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an error in
your SQL syntax; check the manual that corresponds to your MySQL server version
for the right syntax to use near 's JDBC?')' at line 1
```

`PreparedStatement` can handle this situation very easily, as it treats the entire parameter as input. So, the example given using the `PreparedStatement` will work correctly in this case.

23.19.4SQL statements to call stored procedures

JDBC also allows the calling of stored procedures that are stored in the database server. This is done using the `CallableStatement` object. A `CallableStatement` object is created using the `prepareCall()` method on a `Connection` object.

```
CallableStatement prepareCall(String)
```

The method `prepareCall()` takes a primary string parameter, which represents the procedure to be called, and returns a `CallableStatement` object, which is used to invoke stored procedures if the underlying database supports them. Here is an example:

```
String proCall = "{call changePassword(?, ?, ?)}";
CallableStatement cstmt = con.prepareCall(proCall);
```

The variable `cstmt` can now be used to call the stored procedure `changePassword`, which has three input parameters and no result parameter. Whether the ? placeholders are `IN`, `OUT`, or `INOUT` parameters depends on the definition of the stored procedure `changePassword`.

JDBC API allows the following syntax to call stored procedures:

```
{call procedure-name [(?, ?, ...)]}
```

For example, to call a stored procedure with no parameter and no return type, the following syntax is used:

```
{call procedure-name}
```

The following syntax is used to call a procedure that takes a single parameter:

```
{call procedure-name(?)}
```

If a procedure returns a value, the following syntax is used:

```
{? = call procedure-name (?, ?)}
```

MySQL procedures are not allowed to return values. So, the last format is not allowed in MySQL. The web developer must know what stored procedures are available in the underlying database. Before using any stored procedure, one can use the `supportsStoredProcedures()` method on the `DatabaseMetaData` object to verify if the underlying database supports the stored procedure. If it supports, the description of the stored procedures can be obtained using `getProcedures()` on the `DatabaseMetaData` object. Consider the following procedure created in MySQL.

```
DELIMITER //
CREATE PROCEDURE changePassword(IN loginName varchar(10), IN oldPassword
varchar(10), IN newPassword varchar(10))
BEGIN
  DECLARE old varchar(10);
  SELECT password INTO @old FROM users WHERE login=loginName;
  IF @old = oldPassword THEN
    UPDATE users SET password=newPassword WHERE login=loginName;
  END IF;
END //
DELIMITER ;
```

This procedure changes the password of a specified user in the table `users`. It takes three parameters: the login id of the user whose password has to be changed, the old password, and a new password.

If you are using MySQL command prompt to create a procedure, you may face a problem. Note that the stored procedures use ";" as the delimiter. The default MySQL statement delimiter is also ";". This would make the SQL in the stored procedure syntactically invalid. The solution is to temporarily change the command-line utility delimiter using the following command;

```
DELIMITER //
```

This command `DELIMITER //` is not related to the stored procedure. The `DELIMITER` statement is used to change the standard delimiter (semicolon) to another. In this case, the delimiter is changed to //, so that you can have multiple SQL statements separated by the semicolon inside stored procedure. After the `END` keyword, we use delimiter // to show the end of the stored procedure. The last command `DELIMITER ;` changes the delimiter back to the standard one (semicolon).

The `IN` parameters are passed to a `CallableStatement` object using methods of the form `setXxx()`. For example, `setFloat()` and `setBoolean()` methods are used to pass `float` and `boolean` values, respectively.

The following code segment illustrates how to call this procedure.

```
<!--CallableStatement.jsp-->
<%@page import="java.sql.*"%>
<%
   try {
      new com.mysql.jdbc.Driver();
      String url = "jdbc:mysql://uroy:3306/test";
      Connection con = DriverManager.getConnection(url, "root", "nbuser");
      String proCall = "{call changePassword(?, ?, ?)}";
      CallableStatement cstmt = con.prepareCall(proCall);
      String login = request.getParameter("login");
      String oldPassword = request.getParameter("oldPassword");
      String newPassword = request.getParameter("newPassword");
      cstmt.setString(1, login);
      cstmt.setString(2, oldPassword);
      cstmt.setString(3, newPassword);
      if (cstmt.executeUpdate() > 0) {
         out.println("Password changed successfully");
      } else {
         out.println("Couldn't change password");
      }
      cstmt.close();
      conn.close();
   } catch (Exception e) {
      out.println(e);
   }
```

```
%>
```

Following URL may be used to change the password for the user from pass1 to newPass1:

```
http://127.0.0.1:8080/net/jdbc/CallableStatement.jsp?login=user1&oldPassword=pa
ss1&newPassword=newPass1
```

Make sure that the table was created and populated with values as shown below:

```
String create =   "CREATE TABLE users (               "+
                  "  login varchar(20) primary key,   "+
                  "  password varchar(20)          "+
                  ")";
stmt.executeUpdate(create);

String insert = "INSERT INTO users VALUES('user1','pass1')";
stmt.executeUpdate(insert);
insert = "INSERT INTO users VALUES('user2','pass2') ";
stmt.executeUpdate(insert);
```

The `CallableStatement` object also allows batch update exactly like `PreparedStatement`.

Following is an example:

```
String proCall = "{call changePassword(?, ?, ?)}";
CallableStatement cstmt = con.prepareCall(proCall);

cstmt.setString(1, "user1");
cstmt.setString(2, "pass1");
cstmt.setString(3, "newPass1");
cstmt.addBatch();

cstmt.setString(1, "user2");
cstmt.setString(2, "pass2");
cstmt.setString(3, "newPass2");
cstmt.addBatch();

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

This example illustrates how to use batch update facility to associate two sets of parameters with a `CallableStatement` object.

23.20 Retrieving result

A table of data is represented in the JDBC by the `ResultSet` interface. The `ResultSet` objects are usually generated by executing the SQL statements that query the database. A pointer points to a particular row of the `ResultSet` object at a time. This pointer is called *cursor*. The cursor is positioned before the first row when the `ResultSet` object is generated. To retrieve data from a row of `ResultSet`, the cursor must be positioned at the row. The `ResultSet` interface provides methods to move this cursor.

next()

- This method on the `ResultSet` object moves the cursor to the next row of the result set.
- It returns true/false depending upon whether there are more rows in the result set.

Since the `next()` method returns false when there are no more rows in the `ResultSet` object, it can be used in a while loop to iterate through the result set as follows:

```
String query = "SELECT * from users";
ResultSet rs = stmt.executeQuery(query);
```

```
while(rs.next()) {
   //process it
}
```

The `ResultSet` interface provides reader methods for retrieving column values from the row pointed to by the cursor. These have the form `getXxx()`, where `Xxx` is the name of the data type of the column. For example, if data types are string and int, the name of the reader methods are `getString()` and `getInt()`, respectively.

Values can be retrieved using either the column index or the name of the column. Using the column index, in general, is more efficient. The column index starts from 1. The following example illustrates how to retrieve data from a `ResultSet` object.

```
String query = "SELECT * from users";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
   String login = rs.getString("login");
   String password = rs.getString("password");
   System.out.println(login+"\t"+password);
}
```

23.21 Getting Database information

Sometimes, it is necessary to know the capabilities of DataBase Management System (DBMS) before dealing with it. This is because different DBMSs often provide different features, implement them differently, and also use different data types. Moreover, the driver may also implement additional features on top of the DBMS. The `DatabaseMetaData` interface provides methods to collect comprehensive information about a DBMS. We can discover features a DBMS supports and develop our application accordingly. For example, before creating a table, one may want to know what data types are supported by this DBMS. User may also want to know whether the underlying DBMS supports batch update.

A `DatabaseMetaData` object is obtained using the `getMetaData()` method on the `Connection` object as follows:

```
DatabaseMetaData md = con.getMetaData();
```

We can then use various methods on this `DatabaseMetaData` object to collect the required information about the DBMS. Following is a list of commonly used methods:

`getDatabaseMetaData()`

Returns the `DatabaseMetaData` object, which contains detailed information about the underlying database. Some important methods of `DatabaseMetaData` are

`String getSQLKeywords()`

Returns keywords available

`getDatabaseProductName()`

Returns the name of the manufacturer

`getDatabaseProductVersion()`

Returns the current version

`getDriverName()`

Returns driver used

The following JSP page retrieves most of the MySQL database information.

```
<!--DBMetaData.jsp-->
<%@page import="java.sql.*, java.lang.reflect.*"%>
<%
   new com.mysql.jdbc.Driver();
   String url = "jdbc:mysql://uroy:3306/test";
   Connection con = DriverManager.getConnection(url, "root", "nbuser");
   DatabaseMetaData md = con.getMetaData();
   Method[] methods = md.getClass().getMethods();
   Object[] param = new Object[0];
   out.println("<table border=\"1\">");
   for (int i = 0; i < methods.length; i++) {
      if (methods[i].getParameterTypes().length == 0) {
         if (methods[i].getReturnType() == Boolean.TYPE ||
methods[i].getReturnType() == String.class) {
            out.println("<tr>");
            out.println("<td>"+methods[i].getName() + "</td>");
            try {
              out.println("<td>" + methods[i].invoke(md, param)+"</td>");
            }catch(Exception e) {out.println("<td>" + e+"</td>");}
            out.println("</tr>");
         }
      }
   }
   out.println("</table>");
%>
```

The result of this page is shown in Table 21.9.

23.22 Scrollable and Updatable ResultSet

The result set returned so far by a query can be navigated in one direction (forward). Moreover the data the result sets contain are read-only. Any change to the result set does not affect the actual database.

Result sets can be *scrollable* in the sense that the cursor can be moved backward and forward. Additionally, a result set can be *updatable*, such that any change to the result set reflects in the database immediately. A result set can be scrollable as well as updatable. *Note that scrollable and updatable result sets incur significant overhead. So, such result sets should be created if the underlying application performs scrolling.*

In addition to *scrollability* and *updatability*, another important concept, called *sensitivity*, is defined. Sensitivity broadly answers the following question:

```
Can a result set see the changes that are made to the underlying database?
```

If a result cannot see any changes, it is said to be insensitive. Otherwise, the sensitivity of a result set is defined with respect to the database operation as well as the operating party. For example, a result set is said to be sensitive to update if it can see any update operation made on the underlying database. The sensitivity rules are shown in Table 23.3:

The `createStatement()` and `prepareStatement()` methods take extra parameters that specify the type of result returned by subsequent execution of SQL statements. The prototype of the `createStatement()` method to generate a scrollable and updatable result set is as follows:

```
Statement createStatement(int resultSetType, int resultSetConcurrency)
```

Here, scrollability and updatability are controlled by the parameters `resultSetType` and `resultSetConcurrency`, respectively.

23.22.1Scrollability type

The parameter `resultSetType` can assume the following static integer constants defined in `ResultSet`. Their meaning is as follows:

- `TYPE_FORWARD_ONLY`
  If this constant is used, the cursor starts at the first row and can only move forward.

- `TYPE_SCROLL_INSENSITIVE`
  All cursor positioning methods are enabled; the result set does not reflect changes made by others in the underlying table.

- `TYPE_SCROLL_SENSITIVE`
  All cursor positioning methods are enabled; the result set reflects changes made by others in the underlying table.

The visibility of internal and external changes to scrollable `ResultSet` in Oracle JDBC is shown in Table 21.6. (AQ: Please add a line here citing Table 21.6.)

Table 23.3: Visibility of Internal and External Changes to Scrollable Result Set in Oracle JDBC

Scroll Type		TYPE_FORWARD_ONLY	TYPE_SCROLL_SENSITIVE	TYPE_SCROLL_INSENSITIVE
Internal	DELETE	No	Yes	Yes
	UPDATE	Yes	Yes	Yes
	INSERT	No	No	No
External	DELETE	No	No	No
	UPDATE	No	Yes	No
	INSERT	No	No	No

23.22.2Concurrency Type

The parameter `resultSetConcurrency` can assume the following static integer constants defined in `ResultSet`. The following is a brief description:

- `CONCUR_READ_ONLY`
  The result set is not updatable.

- `CONCUR_UPDATABLE`
  Rows can be added and deleted; columns can be updated and are visible to others.

23.22.3Examples

The following example creates a `Statement` object, whose methods will return scrollable, update insensitive, and read-only result sets.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
```

The `ResultSet` object `rs` is now scrollable, but update insensitive.

Originally, result sets could be navigated only in one direction (forward) and starting at only one position (the first row). In JDBC 2.0, the row pointer can be manipulated as if it were an array

index. Some of the important methods available on the scrollable `ResultSet` object are shown in Table 23.4: The following examples demonstrate how to navigate a scrollable result set:

- Move the cursor forward by one row.
```
rs.next();
//or
rs.relative(1);
```

- Move the cursor backward by one row.
```
rs.previous();
//or
rs.relative(-1);
```

- Set the cursor before the first row.
```
rs.beforeFirst();
```

- Set the cursor after the last row.
```
rs.afterLast();
```

- Set the cursor at the first row (row 1).
```
rs.first();
//or
rs.absolute(1);
```

- Set the cursor at the last row.
```
rs.last();
//or
rs.absolute(-1);
```

- Set the cursor at the second row.
```
rs.absolute(2);
```

- Set the cursor at the second last row.
```
rs.absolute(-2);
```

- Move the cursor forward six rows from the current position.
```
rs.relative(6);
//Sets the cursor after the last row, if it goes beyond the last row
```

- Move the cursor backward four rows from the current position.
```
rs.relative(-4);
//Sets the cursor before the first row, if it goes before the first row
```

Table 23.4: Scrollable ResultSet methods

Method	Description
next()	Advances cursor to the next row
previous()	Moves cursor back one row
first()	Sets cursor to the first row
last()	Sets cursor to the last row
beforeFirst()	Sets cursor just before the first row
afterLast()	Sets cursor just after the last row
absolute(int rowNumber)	Sets the cursor to the specified row number. +ve and –ve numbers indicate positions relative to the position before first row and after last row, respectively.

	For example, 1 and -1 represent first and last row, respectively.
relative(int rows)	Forwards or reverses cursor the specified number of rows relative to the current position. +ve number indicates forwarding and –ve number indicates reversing. For example, relative(1) forwards the cursor one position, which is equivalent to next(). Similarly, relative(-1) moves the cursor one position back and is equivalent to previous(). It throws an SQLException if cursor points before the first row or after the last row.
moveToInsertRow()	Sets the cursor to a special row called “insert row” and remembers the current position before moving the cursor.
moveToCurrentRow()	Sets the cursor to the row from where the cursor was moved to “insert row” using moveToInsertRow().

The following example creates a `Statement` object, whose methods will return scrollable as well as external-update-insensitive and updatable result sets.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
```

To update a row in a database table, the following steps are used:

- Obtain an updatable result set.
- Move the cursor to the row to be updated using positioning methods available on the `ResultSet` object.
- Update the value of one or more columns in that row using the `updateXxx()` method on the `ResultSet` object, where `Xxx` is the data type of the column.
- Finally, update the database table using the `updateRow()` method.

The following example demonstrates how to update, insert, or delete a row from a database table using an updatable result set.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

//Updating existing row
rs.absolute(4);
rs.updateString("password","newPassword");
rs.updateRow();

//inseting new row
rs.moveToInsertRow();
rs.updateString(1, "anik");
rs.updateString(2, "anik123");
rs.insertRow();

//Deleting a row
rs.deleteRow()
```

The following JSP page changes the password of a specified user using the updatable result set.

```
<!--UpdatableResultSetDemo.jsp-->
<%@ page import="java.sql.*" %>
<%
   try {
      String login = request.getParameter("login");
      String oldPassword = request.getParameter("oldPassword");
      String newPassword = request.getParameter("newPassword");
```

```
      new com.mysql.jdbc.Driver();
      String url = "jdbc:mysql://uroy:3306/test";
      Connection con = DriverManager.getConnection(url, "root", "nbuser");
      Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
      String query = "SELECT * FROM users WHERE login='" + login + "'";
      ResultSet rs = stmt.executeQuery(query);  //rs contains one row
      rs.next();                                //set cursor at first row
      String password = rs.getString("password");
      if (password.equals(oldPassword)) {
         System.out.println(password);
         rs.updateString("password", newPassword); //update the password column
         rs.updateRow();                           //update database table
      }
   } catch (Exception e) {out.println(e); }
%>
```

Use following URL to verify the above JSP page:

```
http://127.0.0.1:8080/net/jdbc/UpdatableResultSetDemo.jsp?login=user2&oldPasswo
rd=pass2&newPassword=newPass2
```

The following example populates the table questions by inserting questions into the updatable result set.

```
<!--UpdatableResultSetDemo1.jsp-->
<%@ page import="java.sql.*, java.io.*, java.util.*" %>
<%
   try {
      new com.mysql.jdbc.Driver();
      String url = "jdbc:mysql://uroy:3306/test";
      Connection con = DriverManager.getConnection(url, "root", "nbuser");
      Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
      ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
      BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(application.getRealPath("/")+"/jdbc/question.txt")));
      String line = br.readLine();
      while (line != null) {
         StringTokenizer st = new StringTokenizer(line, ":");
         String qno = st.nextToken();
         String question = st.nextToken();

         rs.moveToInsertRow();
         rs.updateString(1, qno);
         rs.updateString(2, question);
         rs.insertRow();
         line = br.readLine();
      }
      br.close();
   } catch (Exception e) {out.println(e); }
%>
```

Use following URL to verify the above JSP page:

```
http://127.0.0.1:8080/net/jdbc/UpdatableResultSetDemo1.jsp
```

Make sure that the table questions exists. Otherwise, create is using following SQL command:

```
create table questions (
   qno   integer primary key,
   question varchar(100)
);
```

The following example creates a `Statement` object, whose methods will return scrollable as well as external-update-sensitive and updatable result sets.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
```

In this case, if the database table is updated, it is reflected in the result set. Before retrieving data from a row, you should invoke the `refreshRow()` method of the `ResultSet` object so that it contains the updated row. The following JSP page shows how to use an updatable result set.

```
<!--UpdateSensitive.jsp-->
<%@page import="java.sql.*"%>
<%!
   Connection con;
   Statement stmt;
   ResultSet rs;
   String query;
   public void jspInit() {
      try {
         new com.mysql.jdbc.Driver();
         String url = "jdbc:mysql://uroy:3306/test";
         con = DriverManager.getConnection(url, "root", "nbuser");
         Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
         query = "SELECT * FROM users";
         rs = stmt.executeQuery(query);
         System.out.println("loaded");

      }catch(Exception e) {}
   }
%>
<table border="1">
<tr><th>Login name</th><th>Password</th></tr>
<%
   try {
      response.setHeader("Pragma", "no-cache");
      response.setHeader("Cache-Control", "no-cache");
      response.setDateHeader("Expires", -1);

      rs.beforeFirst();
      while(rs.next()) {
         rs.refreshRow();
         out.println("<tr><td>" + rs.getString("login") + "</td>");
         out.println("<td>" + rs.getString("password")+"</td></tr>");
      }
   }catch(Exception e) {out.println(e);}
%>
</table>
```

This JSP page creates the `ResultSet` when this page is requested for the first time. The `ResultSet` is created in the `jspInit()` method. Consequently, it becomes an instance variable. For subsequent requests, it simply uses the `ResultSet` variable.

23.23 Result Set Metadata

The `ResultSetMetaData` object is used to retrieve information about the types and properties of the columns and other meta information about a `ResultSet` object. This is sometime very

useful, if you do not know much about the underlying database table. A `ResultSetMetaData` object is obtained using the `getMetaData()` method on the `ResultSet` object as follows:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
ResultSetMetaData rsmd = rs.getMetaData();
```

It can be used to get some useful information such as number of rows, number of columns, column names, and their type. Following are some commonly used methods on the `ResultSetMetaData` object:

`int getColumnCount()`
Returns the number of columns in the result

`String getColumnName(int)`
Returns the name of a column in a result set. Requires an integer argument indicating the position of the column within the result set

`int getColumnType(int)`
Returns the type of the specified column in the form of `java.sql.Types`

`String getColumnTypeName(int)`
Returns the type of the specified column as a string

`String getColumnClassName(int)`
Returns the fully qualified Java type name of the specified column

`int getPrecision(int)`
Returns the number of decimal positions

`int getScale(int)`
Returns the number of digits after the decimal position

`String getTableName(int)`
Returns the name of the column's underlying table

`int isNullable(int)`
Returns a constant indicating whether the specified column can have a NULL value

The following JSP page shows how to retrieve meta information from a `ResultSet` object.

```
<!--ResultSetMetaData.jsp-->
<%@page import="java.sql.*, java.lang.reflect.*"%>
<%
   try {
      Class.forName("org.gjt.mm.mysql.Driver");
      String url = "jdbc:mysql://uroy:3306/test";
      Connection conn = DriverManager.getConnection(url, "root", "nbuser");
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
      ResultSetMetaData rsmd = rs.getMetaData();
      Object obj[] = new Object[1];
      Method[] methods = rsmd.getClass().getDeclaredMethods();
      out.println("<table border=\"0\"><tr><td>Method Name</td>");
      for(int j = 0; j < rsmd.getColumnCount(); j++)
         out.println("<td>" + rsmd.getColumnName(j+1) + "</td>");
      out.println("</tr>");
      for (int i = 0; i < methods.length; i++) {
      if(Modifier.isPublic(methods[i].getModifiers()))
            if (methods[i].getParameterTypes().length == 1) {
               if(!methods[i].getName().equals("isWrapperFor"))
                  if(!methods[i].getName().equals("unwrap")) {
                     out.print("<tr><td>" + methods[i].getName() + "</td>");
                     for(int j=0;j<rsmd.getColumnCount();j++) {
                        obj[0] = new Integer(j+1);
```

```
                        out.print("<td>" + methods[i].invoke(rsmd,obj) +
"</td>");
                    }
                    out.println("</tr>");
                }
            }
        }
        out.println("<table>");
    }catch(Exception e) {e.printStackTrace();}
%>
```

A sample result for MySQL database is shown in Table 23.5:

Table 23.5: Result Set Metadata

Method Name	**qno**	**question**
isWritable	true	true
isCaseSensitive	false	false
getPrecision	11	100
isNullable	0	1
getTableName	questions	questions
getScale	0	0
isCurrency	false	false
isSearchable	true	true
isSigned	true	false
getColumnType	4	12
getColumnDisplaySize	11	100
getColumnLabel	qno	question
isAutoIncrement	false	false
getCatalogName	test	test
getColumnClassName	java.lang.Integer	java.lang.String
getColumnTypeName	INT	VARCHAR
getSchemaName		
isDefinitelyWritable	true	true
getColumnCharacterSet	US-ASCII	Cp1252
getColumnCharacterEncoding	null	null
isReadOnly	false	false
getColumnName	qno	question

Table 23.6: Database Metadata

Method Name	**Return value**
autoCommitFailureClosesAllResultSets	false
getDriverName	MySQL Connector Java

allProceduresAreCallable	false
allTablesAreSelectable	false
supportsTransactions	true
getDriverVersion	mysql-connector-java-5.1.26 (Revision: ${bzr.revision-id})
getCatalogSeparator	.
getCatalogTerm	database
getNumericFunctions	ABS,ACOS,ASIN,ATAN,ATAN2,BIT_COUNT,CEILING,COS,COT,DEGREES,EXP,FLOOR,LOG,LOG10,MAX,MIN,MOD,PI,POW,POWER,RADIANS,RAND,ROUND,SIN,SQRT,TAN,TRUNCATE
getProcedureTerm	PROCEDURE
getSQLKeywords	ACCESSIBLE,ALGORITHM,ANALYZE,ASENSITIVE,BEFORE,BIGINT,BINARY,BLOB,CALL,CHANGE,CONDITION,COPY,DATA,DATABASE,DATABASES,DAY_HOUR,DAY_MICROSECOND,DAY_MINUTE,DAY_SECOND,DELAYED,DETERMINISTIC,DIRECTORY,DISCARD,DISTINCTROW,DIV,DUAL,EACH,ELSEIF,ENCLOSED,ESCAPED,EXCHANGE,EXCLUSIVE,EXIT,EXPLAIN,EXPORT,FLOAT4,FLOAT8,FLUSH,FORCE,FULLTEXT,GENERAL,HIGH_PRIORITY,HOUR_MICROSECOND,HOUR_MINUTE,HOUR_SECOND,IF,IGNORE,IGNORE_SERVER_IDS,IMPORT,INFILE,INOUT,INPLACE,INT1,INT2,INT3,INT4,INT8,ITERATE,KEYS,KILL,LEAVE,LIMIT,LINEAR,LINES,LOAD,LOCALTIME,LOCALTIMESTAMP,LOCK,LONG,LONGBLOB,LONGTEXT,LOOP,LOW_PRIORITY,MASTER_HEARTBEAT_PERIOD,MAXVALUE,MEDIUMBLOB,MEDIUMINT,MEDIUMTEXT,MIDDLEINT,MINUTE_MICROSECOND,MINUTE_SECOND,MOD,MODIFIES,NO_WRITE_TO_BINLOG,OPTIMIZE,OPTIONALLY,OUT,OUTFILE,PARTITION,PURGE,RANGE,READS,READ_ONLY,READ_WRITE,REBUILD,REGEXP,RELEASE,REMOVE,RENAME,REORGANIZE,REPAIR,REPEAT,REPLACE,REQUIRE,RESIGNAL,RETURN,RLIKE,SCHEMAS,SECOND_MICROSECOND,SENSITIVE,SEPARATOR,SHARED,SHOW,SIGNAL,SLOW,SPATIAL,SPECIFIC,SQLEXCEPTION,SQL_BIG_RESULT,SQL_CALC_FOUND_ROWS,SQL_SMALL_RESULT,SSL,STARTING,STRAIGHT_JOIN,TABLES,TABLESPACE,TERMINATED,TINYBLOB,TINYINT,TINYTEXT,TRIGGER,UNDO,UNLOCK,UNSIGNED,USE,UTC_DATE,UTC_TIME,UTC_TIMESTAMP,VARBINARY,VARCHARACTER,WHILE,X509,XOR,YEAR_MONTH,ZEROFILL
getSchemaTerm	
getStringFunctions	ASCII,BIN,BIT_LENGTH,CHAR,CHARACTER_LENGTH,CHAR_LENGTH,CONCAT,CONCAT_WS,CONV,ELT,EXPORT_SET,FIELD,FIND_IN_SET,HEX,INSERT,INSTR,LCASE,LEFT,LENGTH,LOAD_FILE,LOCATE,LOCATE,LOWER,LPAD,LTRIM,MAKE_SET,MATCH,MID,OCT,OCTET_LENGTH,ORD,POSITION,QUOTE,REPEAT,REPLACE,REVERSE,RIGHT,RPAD,RTRIM,SOUNDEX,SPACE,STRCMP,SUBSTRING,SUBSTRING,SUBSTRING,SUBSTRING,SUBSTRING_INDEX,TRIM,UCASE,UPPER
getSystemFunctions	DATABASE,USER,SYSTEM_USER,SESSION_USER,PASSWORD,ENCRYPT,LAST_INSERT_ID,VERSION
getTimeDateFunctions	DAYOFWEEK,WEEKDAY,DAYOFMONTH,DAYOFYEAR,MO

	NTH,DAYNAME,MONTHNAME,QUARTER,WEEK,YEAR,HOUR,MINUTE,SECOND,PERIOD_ADD,PERIOD_DIFF,TO_DAYS,FROM_DAYS,DATE_FORMAT,TIME_FORMAT,CURDATE,CURRENT_DATE,CURTIME,CURRENT_TIME,NOW,SYSDATE,CURRENT_TIMESTAMP,UNIX_TIMESTAMP,FROM_UNIXTIME,SEC_TO_TIME,TIME_TO_SEC
isCatalogAtStart	true
locatorsUpdateCopy	true
nullsAreSortedAtEnd	false
nullsAreSortedHigh	false
nullsAreSortedLow	true
supportsBatchUpdates	true
supportsConvert	false
supportsGroupBy	true
supportsOuterJoins	true
supportsSavepoints	true
supportsUnion	true
supportsUnionAll	true
usesLocalFiles	false
getIdentifierQuoteString	`
supportsDataDefinitionAndDataManipulationTransactions	false
dataDefinitionIgnoredInTransactions	false
storesLowerCaseQuotedIdentifiers	true
storesMixedCaseQuotedIdentifiers	false
storesUpperCaseQuotedIdentifiers	true
supportsANSI92IntermediateSQL	false
supportsAlterTableWithAddColumn	true
supportsAlterTableWithDropColumn	true
supportsCatalogsInDataManipulation	true
supportsCatalogsInIndexDefinitions	true
supportsCatalogsInProcedureCalls	true
supportsCatalogsInTableDefinitions	true
supportsIntegrityEnhancementFacility	false
supportsMixedCaseQuotedIdentifiers	false
supportsOpenCursorsAcrossCommit	false
supportsOpenCursorsAcrossRollback	false
supportsOpenStatementsAcrossCommit	false
supportsOpenStatementsAcrossRollback	false

supportsSchemasInDataManipulation	false
supportsSchemasInIndexDefinitions	false
supportsSchemasInProcedureCalls	false
supportsSchemasInTableDefinitions	false
supportsSubqueriesInComparisons	true
supportsSubqueriesInQuantifieds	true
supportsTableCorrelationNames	true
dataDefinitionCausesTransactionCommit	true
supportsCatalogsInPrivilegeDefinitions	true
supportsDataManipulationTransactionsOnly	false
supportsDifferentTableCorrelationNames	true
supportsSchemasInPrivilegeDefinitions	false
supportsStoredFunctionsUsingCallSyntax	true
doesMaxRowSizeIncludeBlobs	true
generatedKeyAlwaysReturned	true
getDatabaseProductName	MySQL
getDatabaseProductVersion	5.0.51b-community-nt
getExtraNameCharacters	#@
getSearchStringEscape	\
nullPlusNonNullIsNull	true
nullsAreSortedAtStart	false
storesLowerCaseIdentifiers	true
storesMixedCaseIdentifiers	false
storesUpperCaseIdentifiers	false
supportsANSI92EntryLevelSQL	true
supportsANSI92FullSQL	false
supportsColumnAliasing	true
supportsCoreSQLGrammar	true
supportsCorrelatedSubqueries	true
supportsExpressionsInOrderBy	true
supportsExtendedSQLGrammar	false
supportsFullOuterJoins	false
supportsGetGeneratedKeys	true
supportsGroupByBeyondSelect	true
supportsGroupByUnrelated	true
supportsLikeEscapeClause	true
supportsLimitedOuterJoins	true

supportsMinimumSQLGrammar	true
supportsMixedCaseIdentifiers	false
supportsMultipleOpenResults	true
supportsMultipleResultSets	true
supportsMultipleTransactions	true
supportsNamedParameters	false
supportsNonNullableColumns	true
supportsOrderByUnrelated	false
supportsPositionedDelete	false
supportsPositionedUpdate	false
supportsSelectForUpdate	true
supportsStatementPooling	false
supportsStoredProcedures	true
supportsSubqueriesInExists	true
supportsSubqueriesInIns	true
usesLocalFilePerTable	false
providesQueryObjectGenerator	false
getURL	jdbc:mysql://uroy:3306/test
isReadOnly	false
getUserName	root@uroy
toString	com.mysql.jdbc.JDBC4DatabaseMetaData@1b6c838

23.24 Key Words

Atomic Transaction—A transaction that either does not occur or occurs completely without any interleaving

CallableStatement—Statements that are used to call stored procedures in a database

Connection—A class that encapsulates the session/connection to a specific database

Database Metadata—An object that represents the meta information about a database

DDL Statement—SQL statements typically used to create tables

DCL Statement—SQL statements typically used to control database tables

DML Statement—SQL statements typically used to manipulate tables such as insert and update.

DQL Statement—SQL statements typically used to read values from tables

Driver— A Java class that provides interfaces to a specific database.

JDBC— A Java framework that allows us to access databases through Java programs

PreparedStatement—Pre-compiled statements used to fire parameterized queries

ResultSet—The result of a DQL statement

Result Set Metadata—An object that represents the meta information about a result set.

Scrollable Result Set—A result set whose pointer (cursor) can be moved back and forth

Statement—An interface is used to execute static SQL statements

Stored Procedure—Subroutine written using SQL

Updatable Result Set—A result set that can be used directly to modify original database tables

23.25 Summary

Java DataBase Connectivity (JDBC) allows us to access databases through Java programs. It provides Java classes and interfaces to fire SQL and PL/SQL statements, process results (if any), and perform other operations common to databases.

A Java class that provides interfaces to a specific database is called JDBC driver. JDBC drivers are classified into four categories depending upon the way they work.

The following basic steps are followed to work with JDBC: Loading a Driver, Making a connection and Executing an SQL statement

To work with database, an instance of the driver has to be created and registered with the DriverManager class. Two ways we can do this: using static forName() method of the Java class Class or using static registerDriver() method of the DriverManager class.

Once an instance of the driver is created, we create Connnectio object using getConnection() method of DriverManager class. The Connection interface provides methods for obtaining different statement objects that are used to fire SQL statements via the established connection. The Connection object can be used for other purposes such as gathering database information, and committing or rolling back a transaction.

A simple Statement object is instantiated using the createStatement() method on the Connection object. This Statement object defines the many methods to fire different types of SQL commands on the database.

Instead of a simple Statement object, a PreparedStatement object is useful if users want to execute the same SQL statement repeatedly with different data items.

JDBC also allows the calling of stored procedures that are stored in the database server. This is done using the CallableStatement object. A CallableStatement object is created using the prepareCall() method on a Connection object.

A table of data is represented in the JDBC by the ResultSet interface. The ResultSet objects are usually generated by executing the SQL statements that query the database. A pointer points to a particular row of the ResultSet object at a time. This pointer is called cursor. The cursor is positioned before the first row when the ResultSet object is generated. To retrieve data from a row of ResultSet, the cursor must be positioned at the row.

The DatabaseMetaData interface provides methods to collect comprehensive information about a DBMS. We can discover features a DBMS supports and develop our application accordingly. For example, before creating a table, one may want to know what data types are supported by this DBMS.

Result sets can be scrollable in the sense that the cursor can be moved backward and forward. Additionally, a result set can be updatable, such that any change to the result set reflects in the database immediately.

The ResultSetMetaData object is used to retrieve information about the types and properties of the columns and other meta information about a ResultSet object. This is sometime very useful, if you do not know much about the underlying database table. A ResultSetMetaData object is obtained using the getMetaData() method on the ResultSet object.

23.26 Web Resources

`http://docs.oracle.com/javase/tutorial/jdbc/basics/`
Lesson: JDBC Basics

`http://docs.oracle.com/javase/tutorial/jdbc/`
JDBC Database Access

`http://www.tutorialspoint.com/jdbc/jdbc_tutorial.pdf`
JDBC Tutorial

`http://www.javatpoint.com/jdbc-tutorial`
JDBC Tutorial

`http://infolab.stanford.edu/~ullman/fcdb/oracle/or-jdbc.html`
Introduction to JDBC

`http://tutorials.jenkov.com/jdbc/index.html`
Java JDBC

`http://www.techmyguru.com/JDBC/`
Java Database Connectivity (JDBC) Overview

23.27 Exercises

23.27.1Objective Type Questions

1. Which packages contain the JDBC classes?
   a) java.db.sql and javax.db.sql
   b) java.jdbc and javax.jdbc
   c) java.db and javax.db
   d) java.sql and javax.sql

2. Which of the following drivers converts JDBC calls into network protocol, to communicate with the database management system directly?
   a) Type 1 driver
   b) Type 2 driver
   c) Type 3 driver

d) Type 4 driver

3. Which of the following types of objects is used to execute parameterized queries?
   a) ParameterizedStatement
   b) Statement
   c) PreparedStatement
   d) All of the above

4. Which of the following methods on the Statement object is used to execute DML statements?
   a) executeInsert()
   b) execute()
   c) executeQuery()
   d) executeDML()

5. Which type of driver makes a JDBC–ODBC bridge?
   a) Type 1 driver
   b) Type 2 driver
   c) Type 3 driver
   d) Type 4 driver

6. What is the meaning of ResultSet.TYPE_SCROLL_INSENSITIVE
   a) This means that the ResultSet is insensitive to scrolling.
   b) This means that the ResultSet is sensitive to scrolling, but insensitive to changes made by others.
   c) This means that the ResultSet is insensitive to scrolling and insensitive to changes made by others.
   d) This means that the ResultSet is sensitive to scrolling, but insensitive to updates, i.e. not updateable.

7. Which of the following SQL keywords is used to read data from a database table?
   a) SELECT
   b) CHOOSE
   c) READ
   d) EXTRACT

8. Which of the following statement objects is used to call a stored procedure in JDBC?
   a) Statement
   b) PreparedStatement
   c) CallableStatement
   d) ProcedureStatement

9. Which of the following objects is used to obtain the DatabaseMetaData object?
   a) Driver
   b) DriverManager
   c) Connection
   d) ResultSet

10. Which of the following methods is used to call a stored procedure in the database?
    a) execute()

b) executeProcedure()
c) call()
d) run()

11. What will be the effect if we call deleteRow() method on a ResultSet object?
    a)The row pointed to by the cursor is deleted from the ResultSet, but not from the database.
    b)The row pointed to by the cursor is deleted from the ResultSet and from the database
    c)The row pointed to by the cursor is deleted from the database, but not from the ResultSet.
    d)None of the above

12. If you want to work with a ResultSet, which of these methods will not work on PreparedStatement?
    a) execute()
    b) executeQuery()
    c) executeUpdate()
    d) All of the above

13. Which of the following characters is used as a placeholder in CallableStatement?
    a) $
    b) @
    c) ?
    d) #

14. Which one of the following will not get the data from the first column of ResultSet rs, returned from executing the SQL statement: "SELECT login, password FROM USERS"?
    a) rs.getString(0)
    b) rs.getString("login")
    c) rs.getString(1)
    d) All of the above

15. Which of the following interfaces is used to control transactions?
    a) Statement
    b) Connection
    c) ResultSet
    d) DatabaseMetaData

16. Which one of the following represents the correct order?
    a) INSERT, INTO, SELECT, FROM, WHERE
    b) SELECT, FROM, WHERE, INSERT, INTO
    c) INTO, INSERT, VALUES, FROM, WHERE
    d) INSERT, INTO, WHERE, AND, VALUES

17. Which of the following is *not* a benefit of using JDBC?
    a) JDBC programs are tightly integrated with the server operating system.
    b) Systems built with JDBC are relatively easy to move to different platforms.
    c) JDBC programs can be written to connect with a wide variety of databases.
    d) JDBC programs are largely independent of the database to which they are connected.

18. In which of the following layers of the JDBC architecture does the JDBC–ODBC bridge reside?

a) database layer
b) client program layer
c) both client program and database layers
d) JDBC layer

19. Which database application model would an enterprise-wide solution most likely adopt?
    a) The monolithic model
    b) The two-tier model
    c) The three-tier model
    d) The n-tier model

20. Which code segment could execute the stored procedure "calculate()" located in a database server?
    a) Statement stmt = connection.createStatement();
       stmt.execute("calculate()");
    b) CallableStatement cs = con.prepareCall("{call calculate}");
       cs.executeQuery();
    c) PrepareStatement pstmt = connection.prepareStatement("calculate()");
       pstmt.execute();
    d) Statement stmt = connection.createStatement();
       stmt.executeStoredProcedure("calculate()");

23.27.2Subjective Type Questions

1. Write the differences between Type 2 and Type 3 drivers.
2. How do you to get the ResultSet of a stored procedure ?
3. What is the purpose of the setAutoCommit( ) method on the Connection object?
4. How do you move the cursor in scrollable resultsets?
5. Describe the procedure we use to retrieve data from the ResultSet.
6. What are the three statements in JDBC and the differences between them?
7. How do you update a ResultSet programmatically?
8. Why do we use PreparedStatement instead of Statement?
9. What is stored procedure? How do you create a stored procedure?
10. How do you insert and delete a row programmatically?
11. What are batch updates? How are they useful?
12. What are the four types of JDBC driver?
13. How can you use PreparedStatement ?
14. Explain different types of JDBC drivers with their working principle.
15. What are two-tier and three-tier JDBC architecture?
16. Write basic steps to be followed to work with JDBC.
17. Write different types of SQL statements that can be created in JDBC with their functionlaity
18. What does execute() method do? When should we use it?
19. Briefly describe how to make a set of transactions atomic?
20. What are advantages of pre-compiled statements over simple statements?

21. Describe briefly how to call stored procedures in JDBC.
22. What you mean by database metadata? Hot to get them?
23. What are scrollable and updatable result sets?
24. What you mean by result set metadata? Hot to get them?