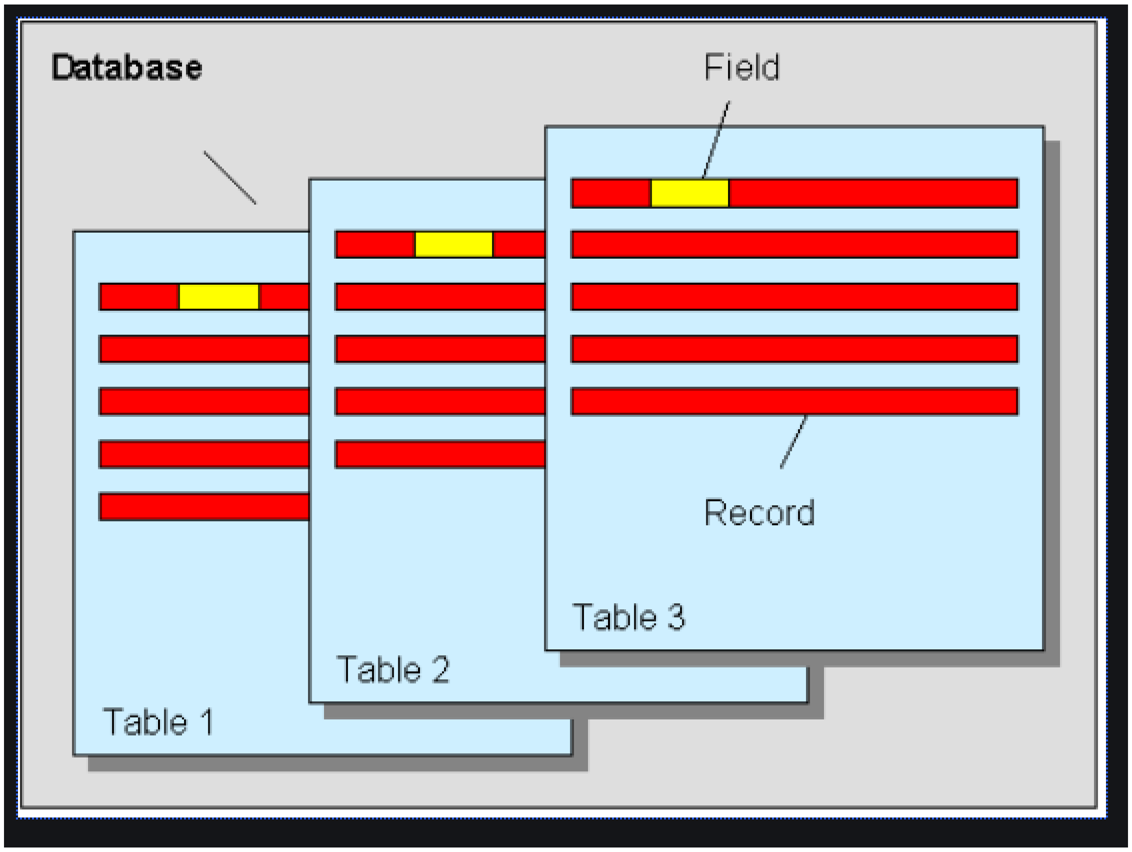
Module 5

The Concept of JDBC; JDBC Driver Types; JDBC Packages; A Brief Overview of the JDBC process; Database Connection; Associating the JDBC/ODBC Bridge with the Database; Statement Objects; ResultSet; Transaction Processing; Metadata, Data types; Exceptions.



J2EE application saves, retrieves and manipulates information stored in a database using web services provided by a J2EE component.

A J2EE component supplies database access using Java data objects contained in the JDBC Application Programming Interface (API).

Java data objects have methods that open a connection to a DBMS and then transmit messages to insert, retrieve, modify or delete data stored in a database.

DBMS uses the same connection to send messages back to the J2EE component.

These messages contain rows of data requested by the J2EE component or information indicating the status of the query being processed by the DBMS. Additional Java data objects are used to interact with data that is returned to the J2EE component by the DBMS.

18. Explain the characteristic nature of a JDBC driver.

The Concept of JDBC
The standard databases are Oracle, DB2, Sybase.

The JDBC driver has to translate the low-level proprietary DBMS messages to low-level messages understood by the JDBC API and vice-versa.

Java programmers could confine to high-level Java data objects defined in the JDBC API to write a routine that interacted with DBMS.

Java data objects convert the routine into low-level messages that conform to the JDBC driver specification and send them to the JDBC driver. The JDBC driver translates the routine into low-level messages that are understood and processed by the DBMS.

JDBC drivers created by DBMS manufacturers have to
1. Open a connection between the DBMS and the J2EE component

2. Translate a low-level equivalent of SQL statements sent by the J2EE component into messages that can be processed by the DBMS.

3. Return data that conforms to the JDBC specification to the JDBC driver.

4. Return information such as error messages that conforms to the JDBC specification to the JDBC driver.

5. Provide transaction management routines that conform to the JDBC specification.

6. Close the connection between the DMS and the J2EE component.

JDBC driver makes J2EE components database independent, which upholds Java's goal of platform independence.

Java code independence is also extended to implementation of the SQL queries. SQL queries are passed from the JDBC API through the JDBC driver to the DBMS without validation. It will be the responsibility of the DBMS to implement SQL statements contained in the query.

JDBC Driver Types https://erainnovator.com/jdbc-drivers/
JDBC driver specification classifies JDBC into 4 groups.

Each group is referred to as a JDBC driver type and addresses a specific need for communicating with various DBMSs.

19. Explain Type-1 and Type-2 JDBC drivers.

Type 1 JDBC-to-ODBC Driver

Microsoft was the first company to devise a way to create a DBMS-independent database program by creating Open Database Connection (ODBC).

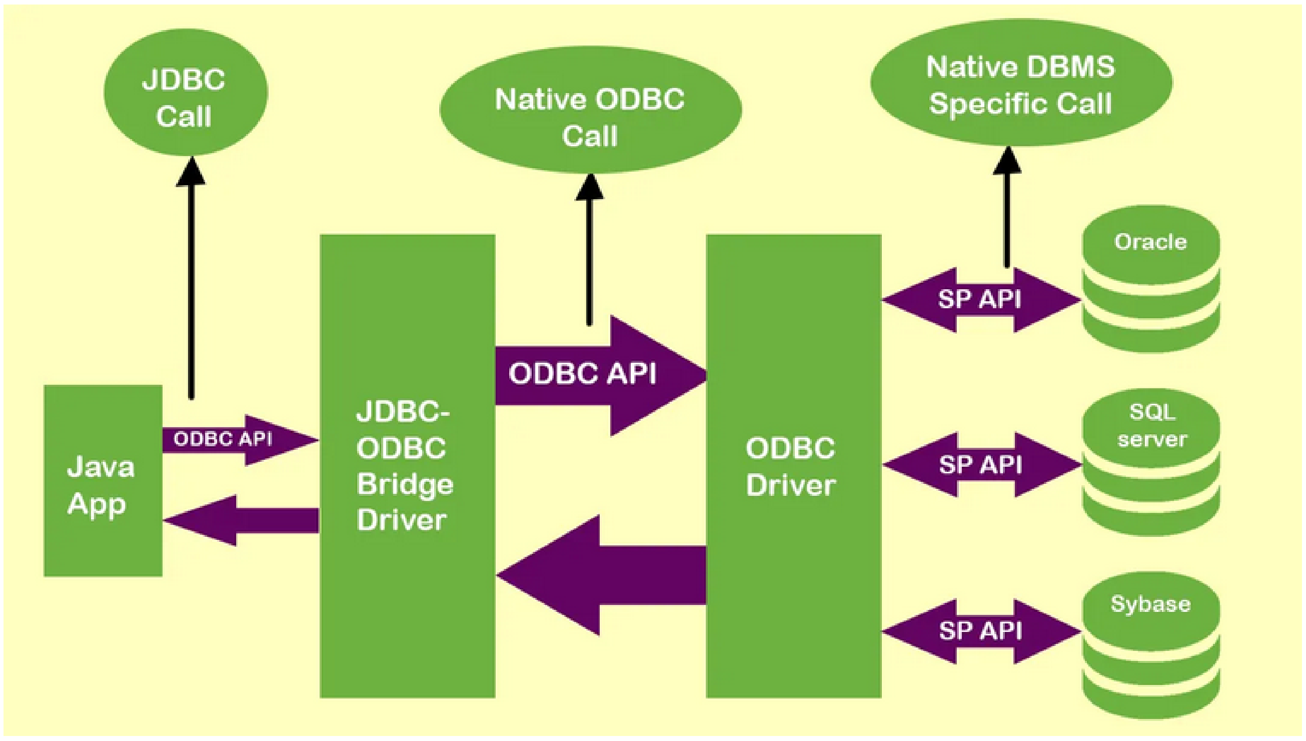
ODBC is the basis from which Sun Microsystems created JDBC.

Both ODBC and JDBC have similar driver specifications and an API.

The JDBC-to-ODBC driver, also called the JDBC/ODBC bridge, is used to translate DBMS calls between the JDBC specification and the ODBC specification.

The JDBC-to-ODBC driver receives messages from a J2EE component that conforms to the JDBC specification. These messages are translated by the JDBC-to-ODBC driver into the ODBC message format, which is then translated into the message format understood by the DBMS.

JDBC/ODBC drivers must not be used in a complex application because the extra translation might negatively impact performance.

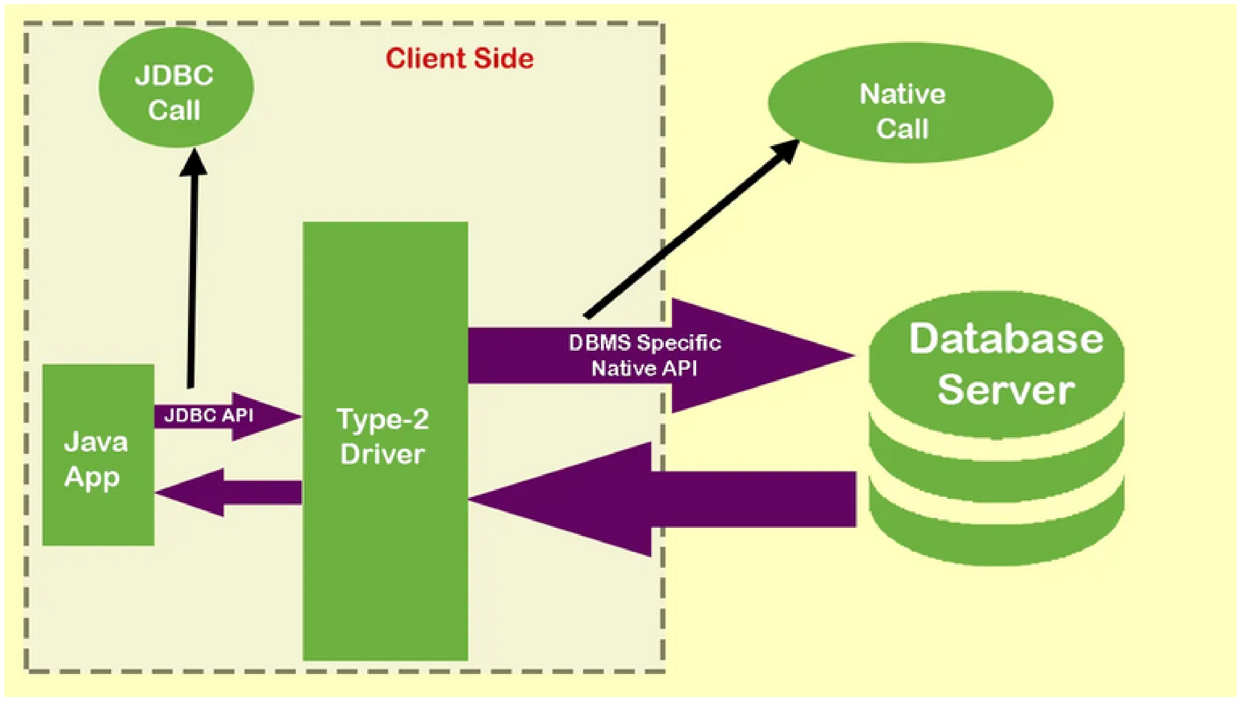


Type 2 Java/Native Code Driver

The Java/Native Code driver uses Java classes to generate platform-specific code-that is code, only understood by a specific DBMS.

The manufacturer of the DBMS provides both the Java/Native Code driver and API classes so the J2EE component can generate the platform-specific code.

The disadvantage of using a Java/Native code driver is the loss of some portability of code. The API classes for the Java/Native Code driver probably will not work with another manufacturer's DBMS.

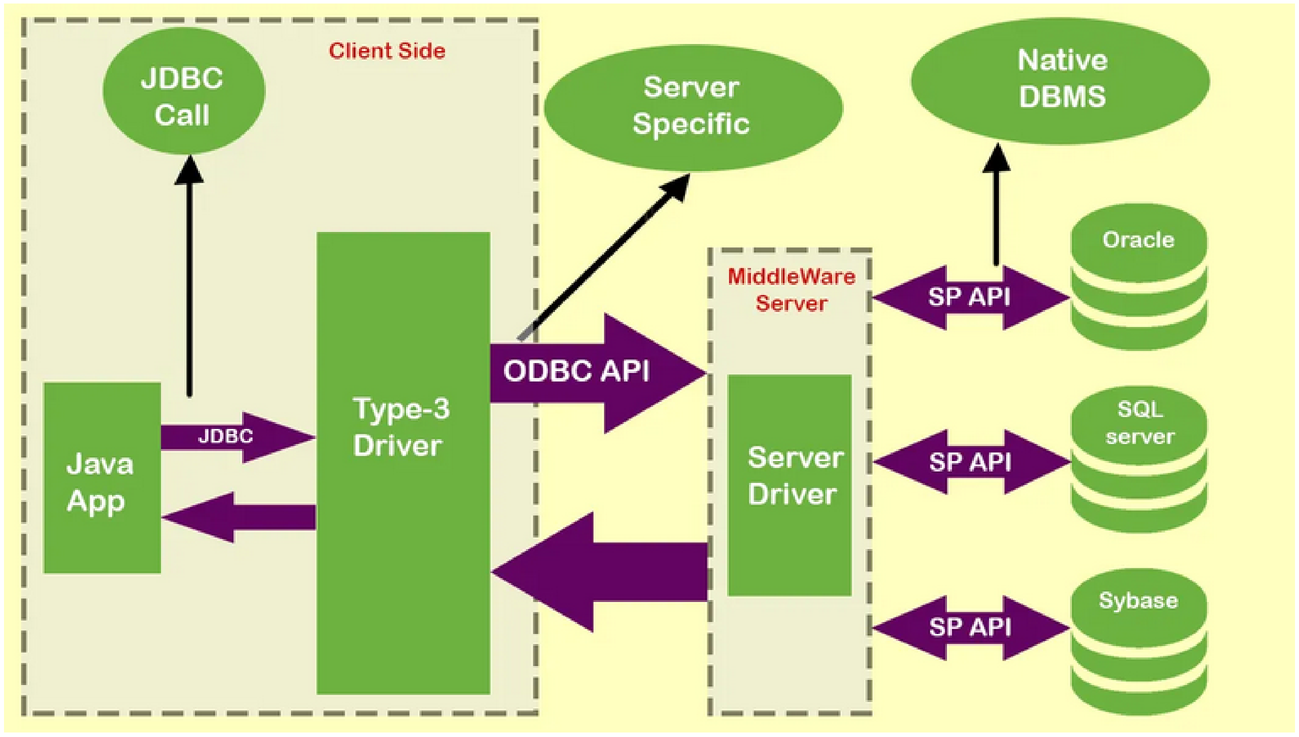


Type 3 JDBC Driver

This is also referred to as the Java Protocol, is the most commonly used JDBC driver.

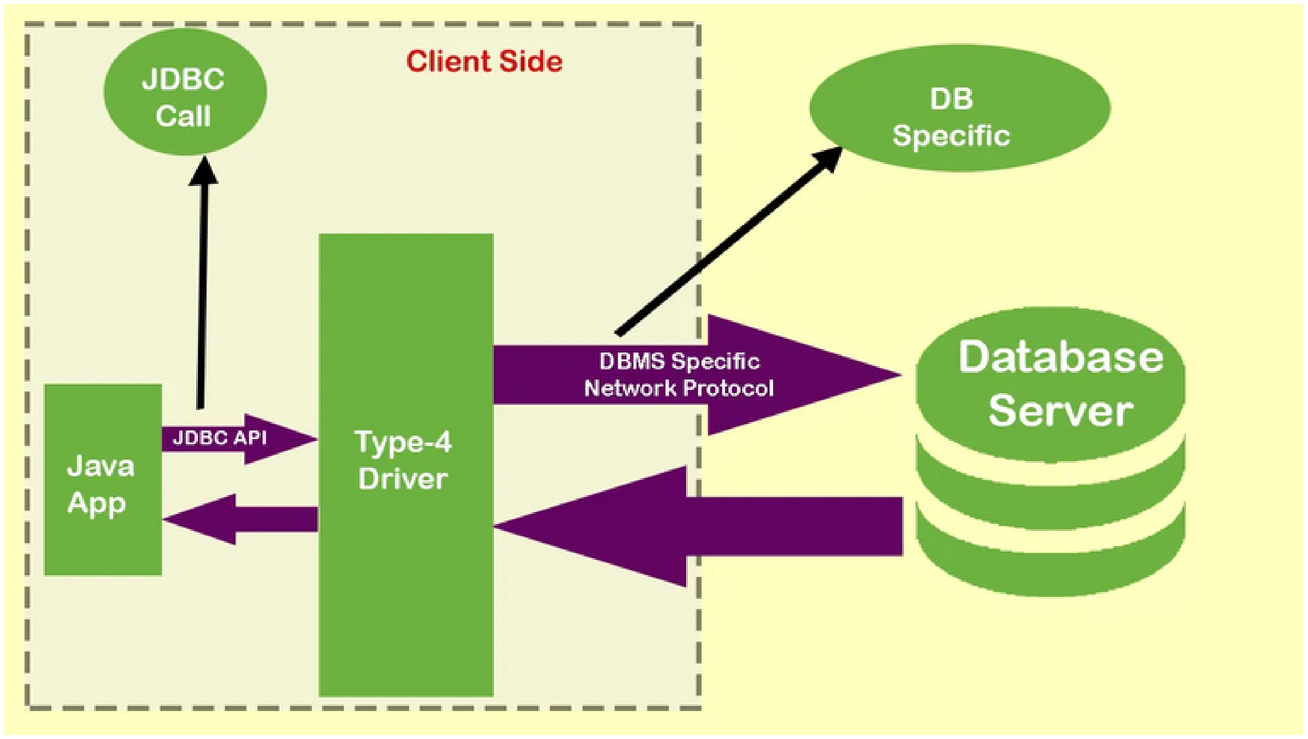
This converts SQL queries into JDBC-formatted statements.

The JDBC-formatted statements are translated into the format required by the DBMS



Type 4 JDBC Driver

This is also termed as Type 4 database protocol. This driver is similar to Type 3 JDBC drivers except SQL queries are translated into the format required by the DBMS. SQL queries need not be converted to JDBC-formatted systems. This is the fastest way to communicate SQL queries to the DBMS.



JDBC Package
The JDBC API is contained in two packages.

The first package is called **java.sql** which contains the **core Java data object of the JDBC API**. These include Java data objects that provide the basics for connecting to the DBMS and interacting with data stored in the DBMS.

java.sql is part of the J2SE. (Java Platform Standard Edition)

The second package that contains JDBC API is **javax.sql**, which extends java.sql and is in the J2EE. (Java Platform Enterprise Edition)
Included in the **javax.sql** package are Java data objects that interact with Java Naming and Directory Interface (JNDI) and Java data objects that manage connection pooling, among other advanced JDBC features.

A Brief Overview of the JDBC Process
J2EE components are (https://www.informit.com/articles/article.aspx?p=28706&seqNum=2)

- Enterprise JavaBeans (EJB)
- Java Servlets
- JavaServer Pages (JSP)
- Java Naming Directory Interface (JNDI)
- Java Database Connectivity (JDBC)
- Java Message Service (JMS)
- Java Transaction API (JTA)
- Java API for XML (JAXP)

Although each J2EE components are different, each uses a similar process for interacting with a DBMS. This process is divided into 4 routines.

1. Loading the JDBC driver
2. Connecting to the DBMS
3. Creating and executing a statement
4. Processing data returned by the DBMS
5. Terminating the connection with the DBMS

Loading the JDBC Driver
JDBC driver must be loaded before the J2EE component connects to the DBMS.

The **Class.forName( )** method is used to load the JDBC driver.
Ex:

A developer working offline creates a J2EE component that interacts with Microsoft Access, the developer must write a routine that loads the JDBC/ODBC Bridge driver called **sun.jdbc.odbc.JdbcOdbcDriver**. The driver is loaded by calling the Class.forName( ) method and passing it the name of the driver as shown below.

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

Connect to the DBMS
Once the driver is loaded, the J2EE component must connect to the DBMS using the **DriverManager.getConnection( )** method.

The **java.sql.DriverManager** class is the highest class in the java.sql hierarchy and is responsible for managing driver information.

The DriverManager.getConnection( ) method is passed the **URL of the database**, **user ID** and **password** if required by the DBMS.

URL is a String object that contains the driver name and the name of the database that is being accessed by the J2EE component.

The DriverManager.getConnection( ) method **returns a Connection interface that is used throughout the process to reference the database**.

public static Connection getConnection(String url) throws SQLException

20. With a suitable code snippet explain how connection can be established with a DBMS using JDBC driver

The java.sql **Connection** interface is another member of the java.sql package that manages communication between the driver and the J2EE component.

It is the java.sql.Connection interface that sends statements to the DBMS for processing.
Ex:

```
import java.sql.*;

public class JDBCExample {
        // JDBC driver name and database URL
        static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
        static final String DB_URL = "jdbc:mysql://localhost/EMP";

         // Database credentials
         static final String USER = "root";
         static final String PASS = "";

        public static void main(String[] args) {
                Connection conn = null;
                Statement stmt = null;
```

```
                    try{
                          //Register JDBC driver
                          Class.forName(JDBC_DRIVER);

                          //Open a connection
                          System.out.println("Connecting to database...");
                          conn = DriverManager.getConnection(DB_URL,USER,PASS);
                          System.out.println(conn);
                     }
                     catch(Exception e){}
             }
     }
```
Output:
```
             com.mysql.cj.jdbc.ConnectionImpl@68999068
```

For connecting to sun jdbc-odbc driver
```
             String url = "jdbc:odbc:CustomerInformation";
             Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```
Where CustomerInformation is the database name.

Create and Execute SQL Statement

The next step, after the JDBC driver is loaded and connection is successfully made with a particular database managed by the DBMS, is to send a SQL query to the DBMS for processing.

A SQL query consists of a series of SQL commands that direct the DBMS to do actions such as to return rows of data to the J2EE component.

The **Connect.createStatement( )** method is used to create a statement object.

The statement object is then used to execute a query and return a ResultSet object that contains the response from the DBMS, which is usually one or more rows of information requested by the J2EE component.

The query is assigned to a String object, which is passed to the Statement object's **executeQuery( )** method.

Once the ResultSet is received from the DBMS, the **close( )** method is called to terminate the statement.

Ex:

```
// Creating statement
stmt = conn.createStatement(  ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                ResultSet.CONCUR_READ_ONLY);
String sql;
sql = "SELECT * FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
rs.close();
stmt.close();
```

Process Data Returned by the DBMS

The java.sql.ResultSet object is assigned with the results received from the DBMS after the query is processed. The java.sql.ResultSet object consists of methods used to interact with data that is returned by the DBMS to the J2EE component.

Ex:

```
// Import required packages
import java.sql.*;

public class JDBCExample {
     // JDBC driver name and database URL
     static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
     static final String DB_URL = "jdbc:mysql://localhost/EMP";

     //  Database credentials
     static final String USER = "root";
     static final String PASS = "";

     public static void main(String[] args) {
          Connection conn = null;
          Statement stmt = null;
          try{
               //Register JDBC driver
               Class.forName(JDBC_DRIVER);

               //Open a connection
               System.out.println("Connecting to database...");
               conn = DriverManager.getConnection(DB_URL,USER,PASS);
               System.out.println(conn);


          //Execute a query to create statement with required arguments for RS.
     stmt = conn.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_READ_ONLY);
               String sql;
               sql = "SELECT * FROM Employees";
               ResultSet rs = stmt.executeQuery(sql);
```

```
                                    boolean rec = rs.next( );
                                    if (!rec)
                                    { System.out.println("No data returned");        return;        }
                                    else
                                    {
                                     System.out.println("ID\tF-name\tL-name\tAge");
                                     do
                                     {
                                        System.out.print(rs.getInt("id")+"\t"+rs.getString("first")+"\t"
                                                          + rs.getString("last")+"\t"+rs.getInt("age"));
                                            System.out.println("\n");
                                     }
                                     while(rs.next());
                                    }

      //Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
 }catch(Exception se){
      //Handle errors for JDBC
      se.printStackTrace();
 }
}//end main
}//end JDBCExample
```

Considering the code above a J2EE component requested details of the employee table.

The result returned by the DBMS is assigned to the ResultSet object called rs. The first time **next( )** method of rs is called, the ResultSet pointer is positioned at the first row in the ResultSet and returns a boolean value. If false is returned it indicates that no rows are present in the ResultSet.

However, if a true value is returned by the next( ) method indicates that at least one row of data may be present in the ResultSet, which causes the code to enter the do..while loop.

The getString( ) method of the ResultSet object is used to copy the value of a specified column in the current row of the ResultSet to a String object.

The getString( ) method is passed the name of the column in the ResultSet whose content needs to be copied and the getString( ) method returns the value from the specified column.

The number of the column can also be passed as a parameter to the getString( ) method instead of passing the column name. However, this numbering must be used only if the columns are specifically named in the SELECT statement (or SELECT * is used), otherwise the order in which the column appears in the ResultSet may not be known, especially because the table might be reorganized since the table was created and therefore the column might be rearranged.

Ex:
Consider the Employees table as below

Id	First	Last	Age
1	aa	aa	12
2	bb	bb	14
3	cc	cc	45

Further, considering the query statement as “select * from Employees”, Id column will be selected as the 1st column followed by First column and then Last and Age will be the 4th column.

Retrieving information by mentioning column name

```
System.out.print(rs.getInt("id")+"\t"+rs.getString("first")+"\t"
                                        + rs.getString("last")+"\t"+rs.getInt("age"));
```

Retrieving information by mentioning column number

```
System.out.print(rs.getInt(1)+"\t"+rs.getString(2)+"\t"+ rs.getString(3)+"\t"+rs.getInt(4));
```

The do..while( ) loop continues until the next( ) method, called as the conditional argument to the while statement, returns a false, which means the pointer is at the end of the ResultSet.

Terminate the Connection to the DBMS
The connection to the DBMS is terminated by using the close( ) method of the Connection object once the J2EE component is finished accessing the DBMS.

The close( ) method throws an exception if a problem is encountered when disengaging the DBMS. Closing the database connection automatically closes the ResultSet, it is better to close the ResultSet explicitly before closing the connection.
Ex: conn.close( )

Database Connection
A J2EE component will not directly connect to a DBMS, instead it connects with the JDBC driver that is associated with DBMS.

However, before this connection is made, the JDBC driver must be loaded and registered with the DriverManager.

The purpose of loading and registering the JDBC driver is to bring the JDBC driver into the Java Virtual Machine (JVM). The JDBC driver is automatically registered with DriverManager once the JDBC driver is loaded and is therefore available to the JVM and can be used by the J2EE component.

The Class.forName( ) method is used to load the JDBC driver.

The Class.forName( ) method throws a ClassNotFoundException if an error occurs when loading the JDBC driver. Errors can be trapped using the catch block whenever the JDBC driver is being loaded.
Ex:

```
try {
        Class.forname("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch(ClassNotFoundException error) {
  System.err.println("Unable to load the JDBC/ODBC bridge."+error.getMessage());
  System.exit(1);
}
```

The Connection
After the JDBC driver is successfully loaded and registered, the J2EE component must connect to the database. The database must be associated with the JDBC driver, which is usually performed by either the database or system administrator.

The data source that the JDBC component will connect to is defined using the URL format. The URL consists of 3 parts, they are

jdbc which indicates that the JDBC protocol is to be used to read the URL
<subprotocol> which is the JDBC driver name
<subname> which is the name of the database.

The connection to the database is established by using one of 3 **getConnection( )** methods of the **DriverManager** object. The getConnection( ) method requests access to the database from the DBMS.

It is upto the DBMS to grant or reject access. A Connection object is returned by the getConnection( ) method if access is granted, otherwise the getConnection( ) method throws a SQLException.

Sometimes DBMS grants access to a database to anyone without authentication. In this case, the J2EE component uses the getConnection(String url) method. One parameter is passed to the method because the DBMS only needs the database to be identified.
Ex:

```
String url = “jdbc:odbc:CustomerInformation”;
Statement DataRequest;
Connection db;
try  {
        Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
        db = DriverManager.getConnection(url);
}
catch(ClassNotFoundException error) {
        System.err.println(“Unable to load the JDBC/ODBC bridge.”+error);
        System.exit(1);
}
catch(SQLException error) {
        System.err.println(“Cannot connect to the database.” + error);
        System.exit(2);
}
```

Other databases limit access to authorized users and require the J2EE to supply a user ID and password with the request to access the database. In this case the J2EE component uses the getConnection(String url, String user, String password) method.

```
Ex: String url = “jdbc:odbc:CustomerInformation”;
      String userID = “jim”;
      String password = “keogh”;
      Statement DataRequest;
      Connection Db;
      try {
          Class.forName(“sun.jdb.odbc.Jdbc.OdbcDriver”);
          Db = DriverManager.getConnection(rl, userID,password);
     }
     catch(ClassNotFoundException error) {
          System.err.println(“Unable to load the JDBC/ODBC bridge.” + error);
          System.exit(1);
     }
     catch(SQLException error) {
          System.err.println(“Cannot connect to the database.”+error);
          System.exit(1);
     }
```

Some DBMS may require additional information besides user ID and password, before the DBMS grants access to the database. This additional information is referred to as properties and must be associated with a **Properties** object, which is passed to the DBMS as a getConnection( ) parameter.

Contents of properties used to access a database are stored in a text file, the contents of which are defined by the DBMS manufacturer.

The J2EE component uses a **FileInputStream** object to open the file and then uses the Properties object **load( )** method to copy the properties into a properties object.

The third version of the getConnection( ) method passes the Properties object and the URL as parameters to the getConnection( ) method.
Ex:

```
        Connection Db;
        Properties props = new Properties( );
        try {
                FileInputStream propFileStream = new FileInputStream(“DBProps.txt”);
                props.load(propFileStream);
        }
        catch(IOException err) {
                System.err.println(“Error loading propFile”);
        System.exit(1);
    }
    try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Db = DriverManager.getConnection(url, props);
    }
    catch(ClassNotFoundException error) {
        System.err.println("Unable to load the JDBC/ODBC bridge." + error);
        System.exit(1);
    }
    catch(SQLException error) {
        System.err.println("Cannot connect to the database." + error);
        System.exit(1);
    }
```

TimeOut

Competition to use the same database is a common occurrence in the J2EE environment and can lead to performance degradation of a J2EE application.

Ex:

A J2EE application that needs database access requests service from an appropriate J2EE component. In turn, the J2EE component attempts to connect to the database.

DBMS may not respond quickly for a number of reasons, which might include that database connections are not available. Rather than wait for a delayed response from the DBMS, the J2EE component will set a timeout period after which the DriverManager will cease to attempt to connect to the database.

The **public static void DriverManager.getLoginTimeout( )** method is used to retrieve from the DriverManager the maximum time the DriverManager is set to wait until it times out. The DriverManager.getLoginTimeout( ) method returns an int that represents seconds.

Associating the JDBC/ODBC Bridge with the Database

Page 133

Statement Objects
Once a connection to the database is opened, the J2EE component creates and sends a query to access data contained in the database. The query is written using SQL.

One of the 3 types of Statement objects is used to execute the query.

These are,
Statement object, which executes a query immediately.
PreparedStatement, is used to execute a compiled query.
CallableStatement is used to execute stored procedures.

The Statement Object
The Statement object is used whenever a J2EE component needs to immediately execute a query without first having the query compiled.

The Statement object contains the **executeQuery( )** method, which is passed the query as an argument. The query is then transmitted to the DBMS for processing.

The executeQuery( ) method returns one **ResultSet** object that contains rows, columns and metadata that represent data requested by query. The ResultSet object also contains methods that are used to manipulate data in the ResultSet.

The execute( ) method of the Statement object is used when there is a necessity to return multiple results.

Another commonly used method of the Statement object is the **executeUpdate( )** method. This is used to execute queries that contain UPDATE and DELETE SQL statements, which may change the values in rows and may remove a row respectively.

The executeUpdate( ) is used to execute INSERT, UDATE, DELETE and DDL statements.
Ex: same example as in “**Process Data Returned by the DBMS”** in page 9

Ex:
```
    package jdbc123;
    //STEP 1. Import required packages
    import java.sql.*;
    public class JDBCExample {
     // JDBC driver name and database URL
            static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
            static final String DB_URL = "jdbc:mysql://localhost/EMP";
```

```
 //  Database credentials
 static final String USER = "root";
 static final String PASS = "";

public static void main(String[] args) {
          Connection conn = null;
          Statement stmt = null;

          try{
                  //STEP 2: Register JDBC driver
                  Class.forName(JDBC_DRIVER);

                  //STEP 3: Open a connection
                  System.out.println("Connecting to database...");
                  conn = DriverManager.getConnection(DB_URL,USER,PASS);
                  System.out.println(conn);
          }
          catch(ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge."+error);
             System.exit(1);
          }
          catch(SQLException error) {
             System.err.println("cannot connect to the database."+ error);
             System.exit(2);
          }

           try {
         //STEP 4: Execute a query to create a statement with required arguments
         for RS example.
                  System.out.println("Creating statement...");
               String sql= "UPDATE Employees SET age=50 WHERE first='abc'
         ";

         stmt                                                                  =
         conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                             ResultSet.CONCUR_READ_ONLY);
                  int rowsUpdated = stmt.executeUpdate(sql);
                  System.out.println("Rows updated "+rowsUpdated);
                  stmt.close();            conn.close();
```

```
                    }
                        catch(SQLException error) {
                            System.err.println("SQL error."+error);
                            System.exit(3);
                        }
    }
}
```

Above statement illustrates the use of executeUpdate( ) method of the Statement object. The SQL query updates a value in the database rather than requesting that data be returned to the J2EE component.

Declaration of **ResultSet** object is replaced with the declaration of an int called **rowsUpdated**.

The query is changed. The SQL UPDATE command directs the DBMS to update the Employees table of the emp database.

The value of the age column of the Employees table is changed to 50 for all the tuples/records whose first value is "abc".

Finally, the executeUpdate( ) method replaces the executeQuery( ) method and is passed the query. The number of rows that are updated by the query is returned to the executeUpdate( ) method by the DBMS and is then assigned to the rowsUpdated int, which can be used for many purposes within the J2EE component such as sending a confirmation notice to the J2EE application that requested database access.

PreparedStatement Object
A SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object's execution methods is called.

Compiling a query is an overhead that is acceptable if the query is called once. However, the compiling process will be considered as an expensive overhead if the query is executed several times by the same instance of the J2EE component during the same session.

An SQL query can be pre-compiled and executed by using the **PreparedStatement** object. In this case the query is constructed normally, but a question mark is used as a placeholder for a value that is inserted into the query after the query is compiled. It is this value that changes each time the query is executed.
Ex:

```
package jdbc123;

//STEP 1. Import required packages
import java.sql.*;

public class JDBCExample {
 // JDBC driver name and database URL
 static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
 static final String DB_URL = "jdbc:mysql://localhost/EMP";

 //  Database credentials
 static final String USER = "root";
 static final String PASS = "";

public static void main(String[] args) {
        Connection conn = null;    Statement stmt = null;    ResultSet rs = null;
        try{
                //STEP 2: Register JDBC driver
                Class.forName(JDBC_DRIVER);

                //STEP 3: Open a connection
                System.out.println("Connecting to database...");
                conn = DriverManager.getConnection(DB_URL,USER,PASS);
                System.out.println(conn);
        }
        catch(ClassNotFoundException error) {
           System.err.println("Unable to load the JDBC/ODBC bridge."+error);
           System.exit(1);
        }
        catch(SQLException error) {
           System.err.println("cannot connect to the database."+ error);
           System.exit(2);
        }

        try {
//STEP 4: Execute a query to create a statement with required arguments for RS example.
                System.out.println("Creating statement...");
        String sql = "SELECT * FROM Employees WHERE first=? AND age=?";
PreparedStatement pst = conn.prepareStatement(sql,
      ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
```

```
            pst.setString(1, "abc");
            pst.setInt(2, 50);

            rs = pst.executeQuery();

            if (rs.first()==false)
            {
                    System.out.println("No records found\n");
                    System.exit(0);
            }

            rs.first();

            System.out.println("ID\tF-name\tL-name\tAge");
            do {
                  System.out.println(rs.getInt(1)+"\t"+rs.getString(2)+"\t"+
                                                            rs.getString(3)+"\t"+rs.getInt(4));
            }while(rs.next());
            conn.close();
        }
        catch(SQLException error) {
              System.err.println("SQL error."+error);    System.exit(3);
        }
 }
}
```

The query directs the DBMS to return all Employees information where the first is “abc” and age is 50. The query has a question mark, which is a placeholder for the value of the employees-first name, which will be inserted into the precompiled query later in the code.

The **preparedStatement( )** method of the Connection object is called to return the PreparedStatement object. The preparedStatement( ) method is passed the query, which is then precompiled.

The **setxxx( )** method of the PreparedStatement object is used to replace the question mark with the value passed to the setxxx( ) method.

There are a number of setxxx( ) methods available in the PreparedStatement object, each of which specifies the data type of the value that is being passed to the setxxx( ) method.

The setString( ) method is used because the employees 1st column first is of type string.

The setxxx( ) requires two parameters, 1st parameter is an integer that identifies the position of the question mark placeholder and the second parameter is the value that replaces the question mark placeholder.

The executeQuery( ) method of the PreparedStatement object is called. The executeQuery( ) statement does not require a parameter because the query that is to be executed is already associated with the PreparedStatement object.

The advantage of using the PreparedStatement object is that the query is precompiled once and the setxxx( ) method is called as needed to change the specified value of the query without having to recompile the query.

The PreparedStatement object also has an execute( ) method and an executeUpdate( ) method.

The precompiling is performed by the DBMS and is referred to as “late binding”. When the DBMS receives the request, the DBMS attempts to match the query to a previously compiled query. If found, then parameters passed to the query using the setxxx( ) methods are set and the query is executed. If not found then the query is compiled and retained by the DBMS for later use.

The JDBC driver passes two parameters to the DBMS. One parameter is the query and the other is an array of late binding variables. Both binding and compiling is performed by the DBMS. The late binding is not associated with the specific object or code block where the preparedStatement is declared.

CallableStatement
The CallableStatement object is used to call a stored procedure from within a J2EE object. A stored procedure is a block of code and is identified by a unique name.

The type and style of ocde depends on the DBMS vendor and are coded in PL/SQL, Transact-SQL, C or any other programming language. The stored procedure is executed by invoking the name of the stored procedure.

The CallableStatement object uses 3 types of parameters when calling a stored procedure, they are IN, OUT and INOUT.

The IN parameter contains any data that needs to be passed to the stored procedure and whose value is assigned using the setxxx( ) method (ex: setString/setInt/setFloat etc., where setxxx( ) is a generic term to refer to any of these methods).

The OUT parameter contains the value returned by the stored procedures, if any. The OUT parameter must be registered using the **registerOutParameter( )** method and then later received by the J2EE component using the getxxx( ) method.

The INOUT parameter is a single parameter that is used to both pass information to the stored procedure and retrieve information from a stored procedure using the same previous techniques.

```
/* STEPS to execute stored procedure in command prompt
        Click the "shell" button in "XAMPP Control Panel"
        # symbol appears in shell
        # mysql -u root -p   then PRESS ENTER
        Since password is empty PRESS ENTER once again

        MariaDB[(none)]>   this appears in prompt
        MariaDB[(none)]>  use emp   PRESS ENTER
        Database changed
        MariaDB[(emp)]>   call display("abc");    // to call the stored procedure.

        SHOW PROCEDURE STATUS   - is the command to
*/
```

```
/*
        Example:  Stored procedure to accept a string value and to return an integer value.
        DELIMITER //
        CREATE PROCEDURE display
          (  IN con VARCHAR(20),
             OUT p INT
          )
        BEGIN
          SELECT * FROM employees WHERE first = con;
          SET p=1234;
          SELECT @p;
        END //
        DELIMITER ;

        In shell prompt
        > call display("abc",@p);
*/
```

Ex:

```
        package jdbc123;

        //STEP 1. Import required packages
        import java.sql.*;

        public class JDBCExample {
                // JDBC driver name and database URL
                static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
                static final String DB_URL = "jdbc:mysql://localhost/EMP";

                //  Database credentials
                static final String USER = "root";
                static final String PASS = "";

        public static void main(String[] args) {
                Connection conn = null;     Statement stmt = null;     ResultSet rs = null;
                try{
                        //STEP 2: Register JDBC driver
                        Class.forName(JDBC_DRIVER);

                        //STEP 3: Open a connection
                        System.out.println("Connecting to database...");
                        conn = DriverManager.getConnection(DB_URL,USER,PASS);
                        System.out.println(conn);
                }
                catch(ClassNotFoundException error) {
                   System.err.println("Unable to load the JDBC/ODBC bridge."+error);
                   System.exit(1);
                }
                catch(SQLException error) {
                   System.err.println("cannot connect to the database."+ error);
                   System.exit(2);
                 }

                 try {
                        System.out.println("Creating statement...");
                        String sql = "{CALL display(?,?) }";
CallableStatement cst = conn.prepareCall(sql,
         ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
                        int i=0;
                        cst.setString(++i, "abc");
                        cst.registerOutParameter(++i, Types.INTEGER);

                        cst.execute();
                        System.out.println(cst.getInt(2));
                 }
                 catch(SQLException error) {
                         System.err.println("SQL error."+error);
                         System.exit(3);
                  }    }   }
```
The above listing illustrates, to call a stored procedure and retrieve a value returned by the stored procedure.

The first statement in the 2nd try block creates a query that calls the stored procedure display( ). The stored procedure requires parameters that are represented by a question mark placeholder. The first question mark receives a value, which acts as IN parameter. The second question mark acts as OUT parameter, whose value will be retrieved after the execution of the procedure.

Next, the prepareCall( ) method of the Connection object is called and is passed the query. This method returns a CallableStatement object. Since, an OUT parameter is used by the stored procedure, the parameter must be registered using the **registerOutParameter( )** of the CallableStatement object.

The setString( ) method of CallableStatement object is used to set the IN parameter for stored procedure. The first parameter is an integer that represents the number of the parameter, which is 1, which represents the first parameter of the stored procedure. The second parameter to setString( ) is the string value that is passed to the stored procedure.

The registerOutParamter( ) method requires two parameters. The first parameter is an integer that represents the number of the parameter, which is 2, which represents the second parameter of the stored procedure. The second parameter to the registerOutParameter( ) is the data type of the value returned by the stored procedure, which is **Types.INTEGER**.

The execute( ) method of the CallableStatement object is called next to execute the query. The execute( ) method does not require the name of the query because the query is already identified when the CallableStatement object is returned by the prepareCall( ) query method.

After the stored procedure is executed, the getInt( ) method is called to return the value of the specified parameter of the stored procedure, which in this code is 2.

ResultSet

A query is used to update, delete and retrieve information stored in a database. The executeQuery( ) method is used to send the query to the DBMS for processing and returns a ResultSet object that contains data that was requested by the query.

The ResultSet object contains methods that are used to copy data from the ResultSet into a Java collection object or variable for further processing.

Data in a ResultSet object is logically organized into a virtual table consisting of rows and columns. In addition to data, the ResultSet object also contains metadata such as column names, column size, and column data type.

The ResultSet uses a virtual cursor to point to a row of the virtual table. A J2EE component must move the virtual cursor to each row and then use other methods of the ResultSet object to interact with the data stored in columns of that row.

The virtual cursor is positioned above the first row of data when the ResultSet is returned by the executeQuery( ) method. The virtual cursor has to be moved to the first row using the next( ) method.

The next( ) method returns a boolean true if the row contains data, otherwise, a boolean false is returned indicating that no more rows exist in the ResultSet.

Once the virtual cursor points to a row, the getxxx( ) method is used to copy data from the row to a collection, object or variable. getxxx( ) method is data type specific. The data type of getxxx( ) method must be the same data type as the column in the ResultSet.

The getxxx( ) method requires one parameter, which is an integer that represents the number of the column that contains the data.
Ex: getString(1) copies the data from the first column of the ResultSet.

Columns appear in the ResultSet in the order in which column names appeared in the SELECT statement in the query.

Ex: consider the SQL query SELECT CustomerFirstName, CustomerLastName FROM Customer. This query directs the DBMS to return two columns. The first column contains customer-first-name and the second column contains customer-last-name. Therefore getString(1) returns data in the customer-first-name column of the current row in the ResultSet.

Reading The ResultSet
Ex: Page 18 code from PreparedStatement Object
Above listing illustrates a commonly used routine to read values from ResultSet into variables that can later be further processed by the J2EE component.

Once a successful connection is made to the database, a query is defined in the second try{ } block to retrieve the all rows from Employees table.

The first( ) method of the ResultSet is called to move the virtual pointer to the first row in the ResultSet. The first( ) method returns true if there is at least one row in the ResultSet and returns false, if no rows are present.

The false value will be trapped by the if statement, where a suitable message is displayed and program terminates.

A true value causes the program to enter the do..while statement, where the getString( ) method is called to retrieve values from 1st to 4th column from the table.

Scrollable ResultSet
The virtual cursor could only be moved down the ResultSet object prior to JDBC 2.1 API.

But the virtual cursor can be moved backwards or even positioned at a specific row, in the recent versions of JDBC. A particular set of rows can be returned in JDBC 2.1 API.

There are 6 methods of the ResultSet object that are used to position the virtual cursor in addition to the **next( )** method, they are **first( ), last( ), previous( ), absolute( ), relative( ) and getRow( )**

The first( ) method moves the virtual cursor to the first row in the ResultSet.
The last( ) method moves the virtual cursor to the last row in the ResultSet.
The previous( ) method moves the virtual cursor to the previous row.
The absolute( ) method positions the virtual cursor at the row number specified by the integer passed as a parameter to it.
The relative( ) method moves the virtual cursor to the specified number of rows contained in the parameter. The parameter is a positive or negative integer where the sign represents the direction the virtual cursor is moved.
Ex: a -4 value to relative( ) method moves the virtual cursor back 4 ros from the current row.

The getrows( ) method returns an integer that represents the number of the current row in the ResultSet.

The Statement object is created using the createStatement( ) of the Connection object, which is essential to handle a scrollable ResultSet by passing the createStatement( ) method one of 3 constants.

These constants are TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE.

The TYPE_FORWARD_ONLY constant restricts the virtual cursor to downward movement, which is the default setting.

The TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE constants permit the virtual cursor to move in both directions.

The TYPE_SCROLL_INSENSITIVE constant makes the ResultSet insensitive to changes made by **any other J2EE component** to data in the table whose rows are reflected in the ResultSet.

The TYPE_SCROLL_SENSITIVE constant makes the ResultSet sensitive to those changes.

Ex:

```
package jdbc123;

//STEP 1. Import required packages
import java.sql.*;

public class JDBCExample {
    // JDBC driver name and database URL
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/EMP";

    //  Database credentials
    static final String USER = "root";
    static final String PASS = "";

    public static void main(String[] args) {
    Connection conn = null;
    ResultSet rs = null;
    try{
        //STEP 2: Register JDBC driver
        Class.forName(JDBC_DRIVER);

        //STEP 3: Open a connection
        System.out.println("Connecting to database...");
        conn = DriverManager.getConnection(DB_URL,USER,PASS);
        System.out.println(conn);
    }
    catch(ClassNotFoundException error) {
       System.err.println("Unable to load the JDBC/ODBC bridge."+error);
       System.exit(1);
    }
    catch(SQLException error) {
       System.err.println("cannot connect to the database."+ error);
       System.exit(2);
    }
```

```
                try {
                //STEP 4: Execute a query to create statement with required arguments for RS
                example.
                        System.out.println("Creating statement...");
                        String sql = "SELECT * FROM Employees";
Statement st = conn.createStatement(
              ResultSet.TYPE_SCROLL_INSENSITIVE,
              ResultSet.CONCUR_READ_ONLY);
                        rs =  st.executeQuery(sql);

                  if (!rs.next( ))
                  {
                        System.out.println("No data found");
                        System.exit(4);
                  }
                  rs.first( );
                  System.out.println("First row "+rs.getString(2));
                  rs.last( );
                  System.out.println("Last row "+rs.getString(2));
                  rs.previous();
                  System.out.println("Previous to Last row "+rs.getString(2));
                  rs.absolute(4);
                  System.out.println("absolute(4) row "+rs.getString(2));
                  rs.relative(-2);
                  System.out.println("relative(-2) row "+rs.getString(2));
                  rs.relative(2);
                  System.out.println("relative(2) row "+rs.getString(2));
                  }
                 catch(SQLException error) {
                         System.err.println("SQL error."+error);
                         System.exit(3);  }

 }
}
```

Not All JDBC Drivers Are Scrollable

Although the JDBC API contains methods to scroll a ResultSet, some JDBC drivers may not support some or all of these features and therefore are unable to return a scrollable ResultSet.

The following code snippet can be used to check whether the JDBC driver in use supports a scrollable ResultSet or not.

Ex:
```
Statement st = conn.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_READ_ONLY);

DatabaseMetaData meta = conn.getMetaData();
boolean forward,insensitive, sensitive;
forward = meta.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY);
insensitive = meta.supportsResultSetType(
                                    ResultSet.TYPE_SCROLL_INSENSITIVE);
sensitive = meta.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE);

System.out.println("forward "+forward);
System.out.println("insensitive " + insensitive);
System.out.println("sensitive " + sensitive);
```

When the J2EE component requests from the ResultSet, some rows are fetched into the driver and returned at one time. Other times, all rows requested may not be retrieved at the same time. In this case, the driver returns to the DBMS and requests another set of rows that are defined by the fetch size, and then discards the current set of rows. This process continues until the J2EE retrieves all rows.

Although the Statement class has a method for setting maximum rows, the method may not be effective if the driver does not implement it.

The maximum row setting is for rows in the ResultSet and not for the number of rows returned by the DBMS.

Ex: The maximum rows can be set to 100. The DBMS might return 500 rows, but the ResultSet object rejects 400 of them.

The fetch size is set by using the **setFetchSize( )** method. Fetch size method is the are of performance tuning-which is handled by the database administrator or the network engineer.

Ex:
```
String sql = "SELECT * FROM Employees";
Statement st = conn.createStatement (
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                            ResultSet.CONCUR_READ_ONLY);
st.setFetchSize(2);
```

```
rs = st.executeQuery(sql);     rs.first( );     System.out.println(rs.getInt(1));
rs.next( );     System.out.println(rs.getInt(1));
```

Updatable ResultSet

Rows contained in the ResultSet are updatable similar to how rows in a table can be updated.

This task is executed by passing the createStatement( ) method of the Connection object the CONCUR_UPDATABLE constant.

Alternative constant that can be passed is CONCUR_READ_ONLY, to the createStatement( ) method to prevent the ResultSet from being updated.

There are 3 ways in which a ResultSet can be modified. These are updating values in a row, deleting a row and inserting a new row. All of these changes are accomplished by using methods of the Statement object.

Update ResultSet

Once the executeQuery( ) method of the Statement object returns a ResultSet, the updatexxx( ) method is used to change the value of a column in the current row of the ResultSet. The xxx in the updatexxx( ) method is replaced with the data type of the column that is to be updated.

The updatexxx( ) method requires two parameters. The first is either the number or name of the column of the ResultSet that is being updated and the second parameter is the value that will replace the value in the column of the ResultSet.
Ex: rs.updateString("last", "DEE");   or   rs.updateString(1, "DEE");

A value in a column of the ResultSet can be replaced with a NULL value by using the updateNull( ) method. The updateNull( ) method requires one parameter, which is either the number or the name of the column in the current row of the ResultSet.

The **updateRow( )** method is called after all the updatexxx( ) methods are called. The updateRow( ) method changes values in columns of the current row of the ResultSet based on the values of the updatexxx( ) methods.

The changes take effect once the updateRow( ) method is called; however the change only occurs in the ResultSet. The corresponding row in the table remains unchanged until an update query is run.

Ex:
```
try {
    String sql = "SELECT * FROM Employees WHERE first='AAA' ";
   Statement st = conn.createStatement(
                   ResultSet.TYPE_SCROLL_SENSITIVE,
                                              ResultSet.CONCUR_UPDATABLE);
    rs = st.executeQuery(sql);
    boolean records = rs.next( );
    if (!records) {
          System.out.println("No data found");
          System.exit(4);
    }
    //rs.updateNull("last");
    rs.updateString(3, "DEF");  // or   rs.updateString("last", "DEE");
    rs.updateRow();
    rs.close();

  }
  catch(SQLException error) {
          System.err.println("SQL error."+error);
          System.exit(3);  }
 }
```

Delete Row in the ResultSet
The deleteRow( ) method is used to remove a row from a ResultSet.

Sometimes deleting rows from ResultSet will be advantageous when processing the ResultSet because this is a way to eliminate rows from future processing.

The deleteRow( ) method is passed an integer that contains the number of the row to be deleted. A good practice is to use the absolute( ) method to move the virtual cursor to the row in the ResultSet that must be deleted.

However, the value of the row must be examined by the program to assure it is the proper row before the deleteRow( ) method is called. The deleteRow( ) method is then passed a zero integer indicating that the current row must be deleted.

Ex:
```
          try {
         String sql = "SELECT * FROM Employees";
         Statement st = conn.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
         rs = st.executeQuery(sql);
         boolean records = rs.next( );
         if (!records) {
          System.out.println("No data found");
          System.exit(4);
         }
         rs.absolute(5);
         System.out.println("Row to be deleted is " + rs.getString(2));
         rs.deleteRow();
         rs.close();
 }
 catch(SQLException error) {
          System.err.println("SQL error."+error);
          System.exit(3);  }
}
```

Insert Row in the ResultSet
Inserting a row into the ResultSet is accomplished using the same technique as is used to update the ResultSet. That is, the updatexxx( ) method is used to specify the column and value that will be placed into the column of the ResultSet.

The **moveToInsertRow( )** method must be used in order to create a room to insert a new row in the ResultSet object. Then the updatexxx( ) methods must be called on each column to fill up new values (except the null valued columns). Finally **insertRow( )** method must be called which causes a new row to be inserted into the ResultSet having values that reflect the parameters in the updatexxx( ) methods, which also updates the underlying database.
Ex:
```
try {
         String sql = "SELECT * FROM Employees";
         Statement st = conn.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                                             ResultSet.CONCUR_UPDATABLE);
```

```
        rs = st.executeQuery(sql);
        boolean records = rs.next( );
        if (!records) {
         System.out.println("No data found");
         System.exit(4);
        }
        rs.moveToInsertRow(); //without this rs.insertRow( ) will throw an error.
        rs.updateInt(1, 5);
        rs.updateString(2, "EEE");
        rs.updateString(3,"eee");
        rs.updateInt(4, 10);
        rs.insertRow( );

        rs.close();

  }
  catch(SQLException error) {
         System.err.println("SQL error."+error);
         System.exit(3);  }
}
```

Transaction Processing

/******

A **transaction**, in the context of a database, is a logical unit that is independently executed for data retrieval or updates. A transaction is also termed as "unit of work" that is achieved within a database design environment.

In relational databases, database transactions must be atomic, consistent, isolated and durable—summarized as the ACID acronym.

*****/

A database transaction consists of a set of SQL statements, each of which must be successfully completed for the transaction to be completed. If one fails, SQL statements that executed successfully up to that point in the transaction must be rolled back.

A database transaction is not completed until the J2EE component calls the **commit( )** method of the Connection object.

All SQL statements executed prior to the call to the commit( ) method can be rolled back. However, once the commit( ) method is called, none of the SQL statements can be rolled back.

The commit( ) method must be called regardless of whether the SQL statement is part of a transaction or not. However, the commit( ) method will be called automatically, because the DBMS has an AutoCommit feature that is by default set to true.

If a J2EE component is processing a transaction, the AutoCommit feature must be deactivated by calling **setAutoCommit( )** method and passing it a false parameter. Once the transaction is completed, the setAutoCommit( ) method is called again-this time by passing a true parameter.

Ex:
```
 try {
          Statement st = conn.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                                                        ResultSet.CONCUR_UPDATABLE);
          Statement st1 = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                                        ResultSet.CONCUR_UPDATABLE);
          conn.setAutoCommit(false);

          String sql = "UPDATE employees set age=190 where first='AAA'";
          String sql1 = "UPDATE employees set age=190 where first='EEE'";

            st.executeUpdate(sql);
            st1.executeUpdate(sql1);

             sql = "Select first from employees where age=190";
             rs=st.executeQuery(sql);
             rs.first();
            System.out.println("After updating...");
            System.out.println(rs.getString(1));    rs.next( );
            System.out.println(rs.getString(1));

           throw new SQLException( );
           //if the above statement is commented then add the below statement
          //conn.commit( );
          }
          catch(SQLException error) {
                 System.err.println("SQL error."+error);
                 try {
                         System.out.println("Transaction is being rolled back");
                         conn.rollback();
                 }
                 catch(SQLException excep) {
                         System.err.print("SQLException");
                         System.err.println(excep.getMessage());
                 }
 }
```

Output:

```
        Connecting to database…
        com.mysql.cj.jdbc.ConnectionImpl@68999068
        After updating…
        AAA
        EEE
        SQL error.java.sql.SQLException
        com.mysql.cj.jdbc.ConnectionImpl@68999068
        Transaction is being rolled back
```

A transaction may consist of many tasks, some of which may not need to be rolled back if the entire transaction fails.

Ex: Consider s1, s2 and s3 are 3 different SQL statements of a transaction, if s3 fails there might be no such logical necessity to roll back all the 3 tasks.

The J2EE component can control the number of tasks that are rolled back by using savepoints. A savepoint is a virtual marker that defines the task at which the rollback stops.

There can be many savepoints used in a transaction. Each savepoint is identified by a unique name. The savepoint name is then passed to the rollback( ) method to specify the point within the transaction where the rollback is to stop.

Ex:

```
       //in main function scope declare
       Savepoint sp1=null;
        …...
        try {
        Statement st = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                                    ResultSet.CONCUR_UPDATABLE);
        Statement st1 = conn.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                                                    ResultSet.CONCUR_UPDATABLE);
```

```
         conn.setAutoCommit(false);

        String sql = "UPDATE employees set age=390 where first='AAA'";
        String sql1 = "UPDATE employees set age=390 where first='EEE'";


         st.executeUpdate(sql);
         sp1 = conn.setSavepoint("sp1");
         st1.executeUpdate(sql1);

         sql = "Select first from employees where age=390";
        rs=st.executeQuery(sql);
        rs.first();
        System.out.println("After updating...");
        System.out.println(rs.getString(1)); rs.next( );
        System.out.println(rs.getString(1));

        throw new SQLException( );
 }
 catch(SQLException error) {
        System.err.println("SQL error."+error);
                try {
                        System.out.println("Transaction is being rolled back to save point sp1");
                        conn.rollback(sp1);
                        conn.commit();
                }
                catch(SQLException excep) {
                        System.err.print("SQLException");
                        System.err.println(excep.getMessage());
                }
 }
```

Only the row containing first='AAA' its age will be updated to 390, and the row containing first='EEE' will not be updated due save point set and roll back executed.

conn.commit( ) is important after rollback otherwise the transaction executed upto savepoint sp1 will not be updated to database.

Another way to combine SQL statements into a transaction is to batch together these statements into a single transaction and then execute the entire transaction. This can be done by using the **addBatch( )** method of the Statement object.

The addBatch( ) method receives a SQL statement as a parameter and places the SQL statement in the batch.

Once all the SQL statements that comprise the transaction are included in the batch, the **executeBatch( )** method is called to execute the entire batch at the same time. The executeBatch( ) method returns an int array that combines the number of SQL statements that were executed successfully.

The int array is displayed if a **BatchUpdateException** error is thrown during the execution of the batch. The batch can be cleared of SQL statements by using the **clearBatch( )** method. The transaction must be committed using the commit( ) method.

Ex:

```
 try {
   st = conn.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                                                    ResultSet.CONCUR_UPDATABLE);
   conn.setAutoCommit(false);

   String sql = "UPDATE employees set age=390 where first='AAA'";
   String sql1 = "UPDATE employee set age=390 where first='EEEE'";
  //table name is employees purposefully employee has been coded to generate error.

   st.addBatch(sql);
   st.addBatch(sql1);

   int [] update = st.executeBatch();

   conn.commit();

   sql = "Select first from employees where age=390";
    rs=st.executeQuery(sql);
    rs.first();
   System.out.println("After updating...");
   System.out.println(rs.getString(1)); rs.next( );
   System.out.println(rs.getString(1));
  }
```

```
    catch(BatchUpdateException error) {
          System.err.println("Batch error");
          System.err.println("SQL state "+error.getSQLState());
          System.err.println("Message "+error.getMessage());
          System.err.println("Vendor "+error.getErrorCode());
          int [ ] updated = error.getUpdateCounts();
          for(int i=0;i<updated.length;i++)
                    System.out.println(updated[i]);
          SQLException sql = error;
          System.out.println("SQL error"+sql);
          try
          {  st.clearBatch();  }
           catch (SQLException e)
          {e.printStackTrace(); }
  }
  catch(Exception error) {}
 }
```

Output:

```
          Connecting to database...
          com.mysql.cj.jdbc.ConnectionImpl@68999068
          Batch error
          SQL state 42S02
          Message Table 'emp.employee' doesn't exist
          Vendor 1146
          1
          -3
          SQL errorjava.sql.BatchUpdateException: Table 'emp.employee' doesn't exist
```

ResultSet Holdability

Whenever the commit( ) method is called, all ResultSet objects that were created for the transaction are closed. Sometimes a J2EE component needs to keep the ResultSet open even after the commit( ) method is called.

createStatement is overloaded in java

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

ResultSet contents can be controlled, even after calling commit( ) by passing one of two constants to the createStatement( ) method. These constants are HOLD_CURSORS_OVER_COMMIT and CLOSE_CURSORS_AT_COMMIT.

The HOLD_CURSORS_OVER_COMMIT constant keeps ResultSet objects open following a call to the commit( ) method and CLOSE_CURSORS_AT_COMMIT closes ResultSet objects when the commit( ) method is called.

RowSets
The JDBC RowSets object is used to encapsulate a ResultSet for use with Enterprise Java Beans (EJB). A RowSEt object contains rows of data from a table(s) that can be used at a disconnected operation.

That is, an EJB can interact with a RowSet object without having to be connected to a DBMS, which is ideal for a J2EE component.

Auto-Generated Keys
It is common for a DBMS to automatically generate unique keys for a table as rows are generated into the table. The **getGeneratedKeys( )** method of the Statement object is called to return keys generated by the DBMS.

The getGeneratedKeys( ) returns a ResultSet object. **ResultSet.getMetaData( )** method can be used to retrieve metadata relating to the automatically generated key, such as the type and properties of the automatically generated key.

Metadata
Metadata is data about data. A J2EE component can access metadata by using the DatabaseMetaData interface. This interface is used to retrieve information about databases, tables, columns, and indexes among other information about the DBMS.

A J2EE component retrieves metadata about the database by calling the **getMetaData( )** method of the Connection object. This method returns a **DatabaseMetaData** object that contains information about the database and its components.

Once the DatabaseMetaData object is obtained, an assortment of methods contained in the DatabaseMetaData object are called to retrieve specific metadata. Here are some of the more commonly used DatabaseMetaData object methods.

getDatabaseProductName( ) Returns the product name of the database
getUserName( ) Returns the username
getURL( ) Returns the URL of the database
getSchema( ) Returns all the schema names available in this database.
getPrimaryKeys( ) Returns primary keys
getProcedures( ) Returns stored procedure names
getTables( ) Returns names of tables in the database.

ResultSet Metadata
There are 2 types of metadata that can be retrieved from the DBMS.

These are metadata that describes the database and metadata that describes the ResultSet.

Metadata that describes the ResultSet is retrieved by calling the **getMetaData( )** method of the ResultSet object. This returns a ResultSetMetaData object.
Ex:

```
        ResultSetMetaData rm = Result.getMetaData( );
```

Once the ResultSet metadata is retrieved, the J2EE component can call methods of the ResultSetMetaData object to retrieve specific kinds of metadata. More commonly used methods are as follows.
getColumnCount( ) Returns the number of columns contained in the ResultSet.
getColumnName(int number) Returns the name of the column specified by the column number.
getColumnType(int number) Returns the data type of the column specified by the column number.

Data Types
The setxxx( ) and getxxx( ) method are used to set a value of a specific data type and to retrieve a value of a specific data type. The xxx in the names of these methods is replaced with the name of the data type.

SQL Type	**Java Type**	**SQL Type**	**Java Type**
CHAR	String	TINYINT	Byte
VARCHAR	String	SMALLINT	Short

LONGVARCHAR	String	INTEGER	Integer
NUMERIC	java.math.BigDecimal	BIGINT	Long
DECIMAL	java.math.BigDecimal	REAL	float
BIT	Boolean	FLOAT	float

SQL Type	**Java Type**
DOUBLE	double
BINARY	Byte[ ]
VARBINARY	byte[ ]

Exceptions
There are 3 kinds of exceptions that are thrown by JDBC methods. They are SQLExceptions, SQLWarnings and DataTruncation.

SQLExceptions commonly reflect a SQL syntax error in the query and are thrown by many of the methods contained in the java.sql package. This exception is also caused by connectivity issues with the database. This can also be raised when trying to access an object that has been closed.

getNextException( ) method of SQLException object is used to return details about the SQL error or a null if the last exception was retrieved. The **getErrorCode( )** method of the SQLException object is used to retrieve vendor specific error codes.

The SQLWarnings throws warnings received by the Connection from the DBMS. The **getWarnings( )** method of the Connection object retrieves the warnings and the **getNextWarning( )** method of the Connection object retrieves subsequent warnings.

Whenever data is lost due to truncation of the data value, a DataTruncation exception is thrown.