```
import java.sql.*;


public class FirstExample {
   // 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 = "username";
   static final String PASS = "password";

   public static void main(String[] args) {
   Connection conn = null;
   Statement stmt = null;
   try{
      //STEP 2: Register JDBC driver
      Class.forName("com.mysql.jdbc.Driver");

      //STEP 3: Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL,USER,PASS);

      //STEP 4: Execute a query
      System.out.println("Creating statement...");
      stmt = conn.createStatement();
      String sql;
      sql = "SELECT id, first, last, age FROM Employees";
      ResultSet rs = stmt.executeQuery(sql);

      //STEP 5: Extract data from result set
      while(rs.next()){
         //Retrieve by column name
         int id  = rs.getInt("id");
```

```
         int age = rs.getInt("age");
         String first = rs.getString("first");
         String last = rs.getString("last");

         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      //STEP 6: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
```

memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver:

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows:

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
catch(IllegalAccessException ex) {
   System.out.println("Error: access problem while loading!");
   System.exit(2);
catch(InstantiationException ex) {
   System.out.println("Error: unable to instantiate driver!");
   System.exit(3);
}
```


Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver:

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
   DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods:

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName

For a better understanding, we suggest you to study our JDBC - Sample Code tutorial.

```
}//end JDBCExample
```

Now let us compile the above example as follows:

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result:

```
C:\>java JDBCExample
Connecting to database...
Creating statement...
Return value is : false
Rows impacted : 1
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 30, First: Sumit, Last: Mittal
Goodbye!
C:\>
```


The PreparedStatement Object

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
```

```
END $$


DELIMITER ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each:

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure:

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
```

```
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end JDBCExample
```


Now let us compile the above example as follows:

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result:

```
C:\>java JDBCExample
Connecting to database...
Creating statement...
Executing stored procedure...
Emp Name with ID:102 is Zaid
Goodbye!
```

9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**public void moveToInsertRow() throws SQLException** Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	**public void moveToCurrentRow() throws SQLException** Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing.

For a better understanding, let us study Navigate - Example Code as discussed below.

Navigate - Example Code

Following is the example, which makes use of few navigation methods described in the Result Set tutorial.

This sample code has been written based on the environment and database setup done in the previous chapters.

Copy and past the following example in JDBCExample.java, compile and run as follows:


```
//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 = "username";
   static final String PASS = "password";

public static void main(String[] args) {
```

```
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);

      // Move cursor to the first row.
      System.out.println("Moving cursor to the first row...");
      rs.first();

      //STEP 6: Extract data from result set
      System.out.println("Displaying record...");
      //Retrieve by column name
      id  = rs.getInt("id");
      age = rs.getInt("age");
      first = rs.getString("first");
      last = rs.getString("last");

      //Display values
      System.out.print("ID: " + id);
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);
      // Move cursor to the first row.

      System.out.println("Moving cursor to the next row...");
      rs.next();

      //STEP 7: Extract data from result set
      System.out.println("Displaying record...");
      id  = rs.getInt("id");
      age = rs.getInt("age");
      first = rs.getString("first");
      last = rs.getString("last");

      //Display values
```

Now let us compile the above example as follows:

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result:

```
C:\>java JDBCExample
Connecting to database...
Creating statement...
Moving cursor to the last...
Displaying record...
ID: 103, Age: 30, First: Sumit, Last: Mittal
Moving cursor to the first row...
Displaying record...
ID: 100, Age: 18, First: Zara, Last: Ali
Moving cursor to the next row...
Displaying record...
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
Goodbye!
C:\>
```


Updating a Result Set:

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type:

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods:

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException**

```
      // required arguments for RS example.
      System.out.println("Creating statement...");
      Statement stmt = conn.createStatement(
                           ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);
     //STEP 5: Execute a query
      String sql = "SELECT id, first, last, age FROM Employees";
      ResultSet rs = stmt.executeQuery(sql);

      System.out.println("List result set for reference....");
      printRs(rs);

      //STEP 6: Loop through result set and add 5 in age
      //Move to BFR postion so while-loop works properly
      rs.beforeFirst();
      //STEP 7: Extract data from result set
      while(rs.next()){
         //Retrieve by column name
         int newAge = rs.getInt("age") + 5;
         rs.updateDouble( "age", newAge );
         rs.updateRow();
      }
      System.out.println("List result set showing new ages...");
      printRs(rs);
      // Insert a record into the table.
      //Move to insert row and add column data with updateXXX()
      System.out.println("Inserting a new record...");
      rs.moveToInsertRow();
      rs.updateInt("id",104);
      rs.updateString("first","John");
      rs.updateString("last","Paul");
      rs.updateInt("age",40);
      //Commit row
      rs.insertRow();
```

BIT	boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	byte	setByte	getByte
SMALLINT	short	setShort	getShort
INTEGER	int	setInt	getInt
BIGINT	long	setLong	getLong
REAL	float	setFloat	getFloat
FLOAT	float	setFloat	getFloat
DOUBLE	double	setDouble	getDouble
VARBINARY	byte[ ]	setBytes	getBytes
BINARY	byte[ ]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

Date & Time Data Types

The java.sql.Date class maps to the SQL DATE type, and the java.sql.Time and java.sql.Timestamp classes map to the SQL TIME and SQL TIMESTAMP data types, respectively.

Following example shows how the Date and Time classes format the standard Java date and time values to match the SQL data type requirements.

```
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.util.*;
public class SqlDateTime {
   public static void main(String[] args) {
      //Get standard date and time
      java.util.Date javaDate = new java.util.Date();
      long javaTime = javaDate.getTime();
      System.out.println("The Java Date is:" +
             javaDate.toString());

      //Get and display SQL DATE
      java.sql.Date sqlDate = new java.sql.Date(javaTime);
      System.out.println("The SQL DATE is: " +
             sqlDate.toString());
      //Get and display SQL TIME
      java.sql.Time sqlTime = new java.sql.Time(javaTime);
      System.out.println("The SQL TIME is: " +
             sqlTime.toString());
      //Get and display SQL TIMESTAMP
      java.sql.Timestamp sqlTimestamp =
      new java.sql.Timestamp(javaTime);
      System.out.println("The SQL TIMESTAMP is: " +
             sqlTimestamp.toString());
     }//end main
}//end SqlDateTime
```

```
}//end JDBCExample
```

Now, let us compile the above example as follows:

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result:

```
C:\>java JDBCExample
Connecting to database...
Creating statement...
Inserting one row....
Commiting data here....
List result set for reference....
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
ID: 106, Age: 20, First: Rita, Last: Tez
ID: 107, Age: 22, First: Sita, Last: Singh
Goodbye!
C:\>
```


Using Savepoints

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

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

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

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

Copy and past the following example in JDBCExample.java, compile and run as follows:

```
//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 = "username";
   static final String PASS = "password";

public static void main(String[] args) {
   Connection conn = null;
   Statement stmt = null;
   try{
      //STEP 2: Register JDBC driver
      Class.forName("com.mysql.jdbc.Driver");

      //STEP 3: Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL,USER,PASS);

      //STEP 4: Set auto commit as false.
      conn.setAutoCommit(false);

      //STEP 5: Execute a query to delete statment with
      // required arguments for RS example.
      System.out.println("Creating statement...");
      stmt = conn.createStatement();

       //STEP 6: Now list all the available records.
```

printStackTrace( )	Prints the current exception, or throwable, and its backtrace to a standard error stream.
printStackTrace(PrintStreams)	Prints this throwable and it's backtrace to the print stream you specify.
printStackTrace(PrintWriterw)	Prints this throwable and it's backtrace to the print writer you specify.

By utilizing the information available from the Exception object, you can catch an exception and continue your program appropriately. Here is the general form of a try block:

```
try {
   // Your risky code goes between these curly braces!!!
}
catch(Exception ex) {
   // Your exception handling code goes between these
   // curly braces, similar to the exception clause
   // in a PL/SQL block.
}
finally {
   // Your must-always-be-executed code goes between these
   // curly braces. Like closing database connection.
}
```


Example

Study the following example code to understand the usage of **try....catch...finally** blocks.

```
//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";
```

```
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

Try the above example by passing wrong database name or wrong username or password and check the result.

```
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 30, First: Sumit, Last: Mittal
ID: 110, Age: 20, First: Sima, Last: Chug

Displaying available rows...
ID: 95, Age: 20, First: Sima, Last: Chug
ID: 100, Age: 35, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 30, First: Sumit, Last: Mittal
ID: 110, Age: 20, First: Sima, Last: Chug
ID: 200, Age: 30, First: Zia, Last: Ali
ID: 201, Age: 35, First: Raj, Last: Kumar
Goodbye!
C:\>
```

Batching with PrepareStatement Object

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

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

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

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(?, ?, ?, ?)";
```

For a better understanding, let us to study the Batching - Example Code with PrepareStatement object as discussed below.

Batching - Example Code

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

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

This sample code has been written based on the environment and database setup done in the previous chapters.

Copy and past the following example in JDBCExample.java, compile and run as follows:


```
// 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 = "username";
   static final String PASS = "password";

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

13. STORED PROCEDURE

We have learnt how to use **Stored Procedures** in JDBC while discussing the JDBC – Statements chapter. This chapter is similar to that section, but it would give you additional information about JDBC SQL escape syntax.

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE getEmpName
   (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END;
```


NOTE: Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database:

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
   (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END $$
```

JDBC SQL Escape Syntax

The escape syntax gives you the flexibility to use database specific features unavailable to you by using standard JDBC methods and properties.

The general SQL escape syntax format is as follows:

```
{keyword 'parameters'}
```

Here are the following escape sequences, which you would find very useful while performing the JDBC programming:

d, t, ts Keywords

They help identify date, time, and timestamp literals. As you know, no two DBMSs represent time and date the same way. This escape syntax tells the driver to render the date or time in the target database's format. For Example:

```
{d 'yyyy-mm-dd'}
```

Where yyyy = year, mm = month; dd = date. Using this syntax {d '2009-09-03'} is March 9, 2009.

Here is a simple example showing how to INSERT date in a table:

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

//Insert data ==> ID, First Name, Last Name, DOB
String sql = "INSERT INTO STUDENTS VALUES" +
             "(100,'Zara','Ali', {d '2001-12-16'})";


stmt.executeUpdate(sql);
```

Similarly, you can use one of the following two syntaxes, either **t** or **ts**:

```
{t 'hh:mm:ss'}
```

Where hh = hour; mm = minute; ss = second. Using this syntax {t '13:30:29'} is 1:30:29 PM.

```
{ts 'yyyy-mm-dd hh:mm:ss'}
```

This is combined syntax of the above two syntax for 'd' and 't' to represent timestamp.

```
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/";

   //  Database credentials
   static final String USER = "username";
   static final String PASS = "password";

   public static void main(String[] args) {
   Connection conn = null;
   Statement stmt = null;
   try{
      //STEP 2: Register JDBC driver
      Class.forName("com.mysql.jdbc.Driver");

      //STEP 3: Open a connection
      System.out.println("Connecting to a selected database...");
      conn = DriverManager.getConnection(DB_URL, USER, PASS);
      System.out.println("Connected database successfully...");

      //STEP 4: Execute a query
      System.out.println("Deleting database...");
      stmt = conn.createStatement();

      String sql = "DROP DATABASE STUDENTS";
      stmt.executeUpdate(sql);
      System.out.println("Database deleted successfully...");
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
```

```
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";

//  Database credentials
static final String USER = "username";
static final String PASS = "password";

public static void main(String[] args) {
Connection conn = null;
Statement stmt = null;
try{
   //STEP 2: Register JDBC driver
   Class.forName("com.mysql.jdbc.Driver");

   //STEP 3: Open a connection
   System.out.println("Connecting to a selected database...");
   conn = DriverManager.getConnection(DB_URL, USER, PASS);
   System.out.println("Connected database successfully...");

   //STEP 4: Execute a query
   System.out.println("Creating statement...");
   stmt = conn.createStatement();
   String sql = "UPDATE Registration " +
                "SET age = 30 WHERE id in (100, 101)";
   stmt.executeUpdate(sql);

   // Now you can extract all the records
   // to see the updated records
   sql = "SELECT id, first, last, age FROM Registration";
   ResultSet rs = stmt.executeQuery(sql);

   while(rs.next()){
      //Retrieve by column name
      int id  = rs.getInt("id");
```

```
ID: 103, Age: 28, First: Sumit, Last: Mittal
Goodbye!
C:\>
```

```
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            conn.close();
      }catch(SQLException se){
      }// do nothing
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end JDBCExample
```

Now, let us compile the above example as follows:

```
C:\>javac JDBCExample.java
C:\>
```


When you run **JDBCExample**, it produces the following result:

```
C:\>java JDBCExample
Connecting to a selected database...
Connected database successfully...
Creating statement...
Fetching records without condition...
ID: 100, Age: 30, First: Zara, Last: Ali
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
Fetching records with condition...
ID: 100, Age: 30, First: Zara, Last: Ali
```

```
         String first = rs.getString("first");
         String last = rs.getString("last");

         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }

      // Extract records in descending order by first name.
      System.out.println("Fetching records in descending order...");
      sql = "SELECT id, first, last, age FROM Registration" +
                  " ORDER BY first DESC";
      rs = stmt.executeQuery(sql);

      while(rs.next()){
         //Retrieve by column name
         int id  = rs.getInt("id");
         int age = rs.getInt("age");
         String first = rs.getString("first");
         String last = rs.getString("last");

         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      rs.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
```