

Unit-III

JDBC

JDBC Overview – JDBC implementation – Connection class – Statements - Catching Database Results, handling database Queries. Networking– InetAddress class – URL class- TCP sockets – UDP sockets, Java Beans –RMI.

JDBC Overview:

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Why to Learn JDBC?

JDBC stands for **Java Database Connectivity**, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Applications of JDBC

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as –

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas –

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

Audience

This tutorial is designed for Java programmers who would like to understand the JDBC framework in detail along with its architecture and actual usage.

JDBC Implementation:

What is JDBC?

JDBC stands for **Java Database Connectivity**, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as –

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects –

- [Core JAVA Programming](#)
- [SQL or MySQL Database](#)

JDBC Architecture

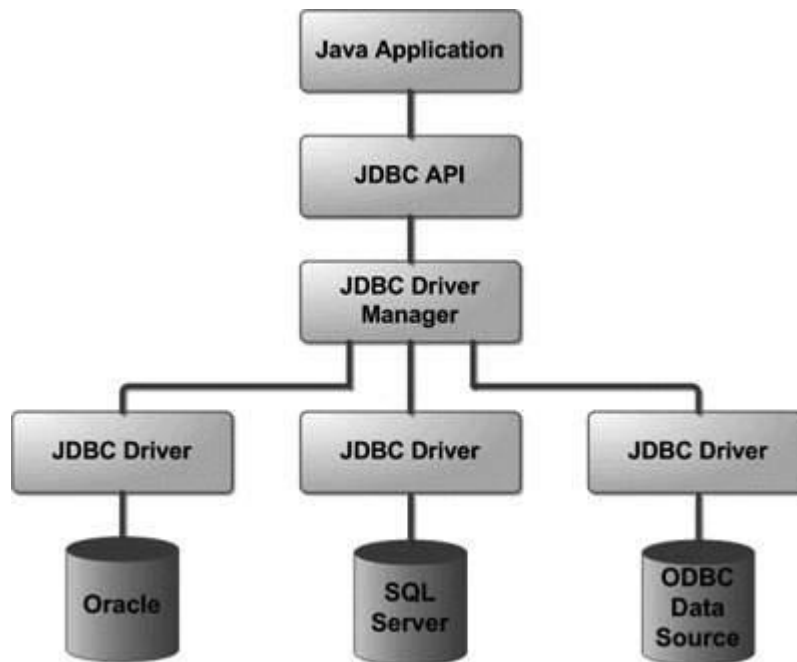
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers –

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application –



Common JDBC Components

The JDBC API provides the following interfaces and classes –

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas –

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.

- Annotations.

Connection class:

Create a DSN-less Database Connection

The easiest way to connect to a database is to use a DSN-less connection. A DSN-less connection can be used against any Microsoft Access database on your web site.

If you have a database called "northwind.mdb" located in a web directory like "c:/webdata/", you can connect to the database with the following ASP code:

```
<%
set conn=Server.CreateObject("ADODB.Connection")
conn.Provider="Microsoft.Jet.OLEDB.4.0"
conn.Open "c:/webdata/northwind.mdb"
%>
```

Note, from the example above, that you have to specify the Microsoft Access database driver (Provider) and the physical path to the database on your computer.

Create an ODBC Database Connection

If you have an ODBC database called "northwind" you can connect to the database with the following ASP code:

```
<%
set conn=Server.CreateObject("ADODB.Connection")
conn.Open "northwind"
%>
```

With an ODBC connection, you can connect to any database, on any computer in your network, as long as an ODBC connection is available.

An ODBC Connection to an MS Access Database

Here is how to create a connection to a MS Access Database:

1. Open the **ODBC** icon in your Control Panel.
2. Choose the **System DSN** tab.
3. Click on **Add** in the System DSN tab.
4. **Select** the Microsoft Access Driver. Click **Finish**.
5. In the next screen, click **Select** to locate the database.
6. Give the database a **Data Source Name (DSN)**.
7. Click **OK**.

Note that this configuration has to be done on the computer where your web site is located. If you are running Personal Web Server (PWS) or Internet Information Server (IIS) on your own computer, the instructions above will work, but if your web site is located on a remote server, you have to have physical access to that server, or ask your web host to do this for you.

The ADO Connection Object

The ADO Connection object is used to create an open connection to a data source. Through this connection, you can access and manipulate a database.

Property	Description
Attributes	Sets or returns the attributes of a Connection object
CommandTimeout	Sets or returns the number of seconds to wait while attempting to execute a command
ConnectionString	Sets or returns the details used to create a connection to a data source
ConnectionTimeout	Sets or returns the number of seconds to wait for a connection to open
CursorLocation	Sets or returns the location of the cursor service
DefaultDatabase	Sets or returns the default database name
IsolationLevel	Sets or returns the isolation level
Mode	Sets or returns the provider access permission
Provider	Sets or returns the provider name
State	Returns a value describing if the connection is open or closed
Version	Returns the ADO version number

Statements:

Once a connection is obtained we can interact with the database. The *JDBC Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's `createStatement()` method, as in the following example –

```
Statement stmt = null;
try {
    stmt = conn.createStatement();
    ...
}
```

```

catch (SQLException e) {
    ...
}
finally {
    ...
}

```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL):** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL):** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```

Statement stmt = null;
try {
    stmt = conn.createStatement();
    ...
}
catch (SQLException e) {
    ...
}
finally {
    stmt.close();
}

```

For a better understanding, we suggest you to study the [Statement - Example tutorial](#).

The PreparedStatement Objects

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) {
    ...
}

```

```
finally {  
    ...  
}
```

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement. The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an `SQLException`.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) `execute()`, (b) `executeQuery()`, and (c) `executeUpdate()` also work with the `PreparedStatement` object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a `Statement` object, for the same reason you should also close the `PreparedStatement` object.

A simple call to the `close()` method will do the job. If you close the `Connection` object first, it will close the `PreparedStatement` object as well. However, you should always explicitly close the `PreparedStatement` object to ensure proper cleanup.

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

For a better understanding, let us study [Prepare - Example Code](#).

The CallableStatement Objects

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

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 {
  ...
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders. Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return. Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

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

Catching database Results:

This chapter provides an example of how to create a simple JDBC application. This will show you how to open a database connection, execute a SQL query, and display the results. All the steps mentioned in this template example, would be explained in subsequent chapters of this tutorial.

Creating JDBC Application

There are following six steps involved in building a JDBC application –

- **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Sample Code

This sample example can serve as a **template** when you need to create your own JDBC application in the future.

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

Copy and paste the following example in FirstExample.java, compile and run as follows –

```
//STEP 1: Import required packages
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
    } //end try
    System.out.println("Goodbye!");
} //end main
} //end FirstExample

```

Now let us compile the above example as follows –

```

C:\>javac FirstExample.java
C:\>

```

When you run **FirstExample**, it produces the following result –

```

C:\>java FirstExample
Connecting to database...
Creating statement...
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
C:\>

```

Handling Database Queries:

The statement interface provides methods that are used to execute static SQL statements. The SQL statements are queries, insertions, updates and deletions, etc.

Some of the methods of the Statement interface are as follows.

- **void close():** This method closes [database](#) connections associated with Statement's object and releases [JDBC](#) resources.
- **boolean execute (String sql):** This method is used to execute an SQL statement that might produce multiple result sets or multiple counts. It returns true if multiple result sets are produced and returns false if multiple counts are generated.

- **int executeUpdate(String sql):** This method is used to execute an SQL statement that gives [information](#) about number of affected rows. For example, if we execute the statements like INSERT, DELETE, the result will only be a number of rows affected.
- **ResultSet executeQuery(String sql):** This method executes a query and returns a Results et object. A result set is just like a table containing the resultant data.
- **ResultSet getResultSet () :** This method returns the first result set or multiple count generated on the execution of execute (String sql) method. If there is no result set, it returns null value.
- **int getUpdateCount():** After the execution of the execute (String sql) method updates counts may be generated. This method returns the first update count and clears it. More update counts can be retrieved by getMoreResults (). Note that this method returns value -1 if there is no update count or when only result sets are generated on execution of execute (String sql) method.
- **boolean getMoreResults () :** This method is used to retrieve the next result set in multiple result set or to the next count in multiple update count generated on execution of execute (String sql) method. It returns a true value if multiple sets are available and returns a false value if multiple update counts are available.

Networking:

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Why to Learn JDBC?

JDBC stands for **Java Database Connectivity**, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Applications of JDBC

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as –

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas –

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

InetAddress Class:

Java InetAddress class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name *for example* www.javatpoint.com, www.google.com, www.facebook.com, etc.

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of InetAddress represents the IP address with its corresponding host name. There are two types of address types: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

Moreover, InetAddress has a cache mechanism to store successful and unsuccessful host name resolutions.

Commonly used methods of InetAddress class

Method	Description
public static InetAddress getByName(String host) throws UnknownHostException	it returns the instance of InetAddress containing LocalHost IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	it returns the instance of InetAddress containing local host name and address.
public String getHostName()	it returns the host name of the IP address.
public String.getHostAddress()	it returns the IP address in string format.

Example of Java InetAddress class

Let's see a simple example of InetAddress class to get ip address of www.javatpoint.com website.

1. **import** java.io.*;
2. **import** java.net.*;
3. **public class** InetDemo{
4. **public static void** main(String[] args){
5. **try**{
6. InetAddress ip=InetAddress.getByName("www.javatpoint.com");
- 7.

```

8. System.out.println("Host Name: "+ip.getHostName());
9. System.out.println("IP Address: "+ip.getHostAddress());
10. }catch(Exception e){System.out.println(e);}
11. }
12. }

```

Output:

```

Host Name: www.javatpoint.com
IP Address: 206.51.231.148

```

URL Class:

Java URL

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

1. <https://www.javatpoint.com/java-tutorial>



A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.
3. **Port Number:** It is an optional attribute. If we write `http://www.javatpoint.com:80/sonoojaiswal/`, 80 is the port number. If port number is not mentioned in the URL, it returns -1.
4. **File Name or directory name:** In this case, index.jsp is the file name.

Constructors of Java URL class

URL(String spec)

Creates an instance of a URL from the String representation.

URL(String protocol, String host, int port, String file)

Creates an instance of a URL from the given protocol, host, port number, and file.

URL(String protocol, String host, int port, String file, URLStreamHandler handler)

Creates an instance of a URL from the given protocol, host, port number, file, and handler.

URL(String protocol, String host, String file)

Creates an instance of a URL from the given protocol name, host name, and file name.

URL(URL context, String spec)

Creates an instance of a URL by parsing the given spec within a specified context.

URL(URL context, String spec, URLStreamHandler handler)

Creates an instance of a URL by parsing the given spec with the specified handler within a given context.

Commonly used methods of Java URL class

The java.net.URL class provides many methods. The important methods of URL class are given below.

Method	Description
public String getProtocol()	it returns the protocol of the URL.
public String getHost()	it returns the host name of the URL.
public String getPort()	it returns the Port Number of the URL.
public String getFile()	it returns the file name of the URL.
public String getAuthority()	it returns the authority of the URL.
public String toString()	it returns the string representation of the URL.
public String getQuery()	it returns the query string of the URL.
public String getDefaultPort()	it returns the default port of the URL.
public URLConnection.openConnection()	it returns the instance of URLConnection i.e. associated with this URL.
public boolean equals(Object obj)	it compares the URL with the given object.
public Object getContent()	it returns the content of the URL.
public String getRef()	it returns the anchor or reference of the URL.
public URI toURI()	it returns a URI of the URL.

Example of Java URL class

```

1. //URLDemo.java
2. import java.net.*;
3. public class URLDemo{
4.     public static void main(String[] args){
5.         try{
6.             URL url=new URL("http://www.javatpoint.com/java-tutorial");
7.
8.             System.out.println("Protocol: "+url.getProtocol());
9.             System.out.println("Host Name: "+url.getHost());
10.            System.out.println("Port Number: "+url.getPort());
11.            System.out.println("File Name: "+url.getFile());
12.
13.        }catch(Exception e){System.out.println(e);}
14.    }
15. }

```

Test it Now

Output:

```

Protocol: http
Host Name: www.javatpoint.com
Port Number: -1
File Name: /java-tutorial

```

Let us see another example URL class in Java.

```
1. //URLDemo.java
2. import java.net.*;
3. public class URLDemo{
4.     public static void main(String[] args){
5.     try{
6.     URL url=new URL("https://www.google.com/search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8");
7.
8.     System.out.println("Protocol: "+url.getProtocol());
9.     System.out.println("Host Name: "+url.getHost());
10.    System.out.println("Port Number: "+url.getPort());
11.    System.out.println("Default Port Number: "+url.getDefaultPort());
12.    System.out.println("Query String: "+url.getQuery());
13.    System.out.println("Path: "+url.getPath());
14.    System.out.println("File: "+url.getFile());
15.
16. }catch(Exception e){System.out.println(e);}
17. }
18. }
```

Output:

```
Protocol: https
Host Name: www.google.com
Port Number: -1
Default Port Number: 443
Query String: q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
Path: /search
File: /search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
```

TCP Sockets:

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

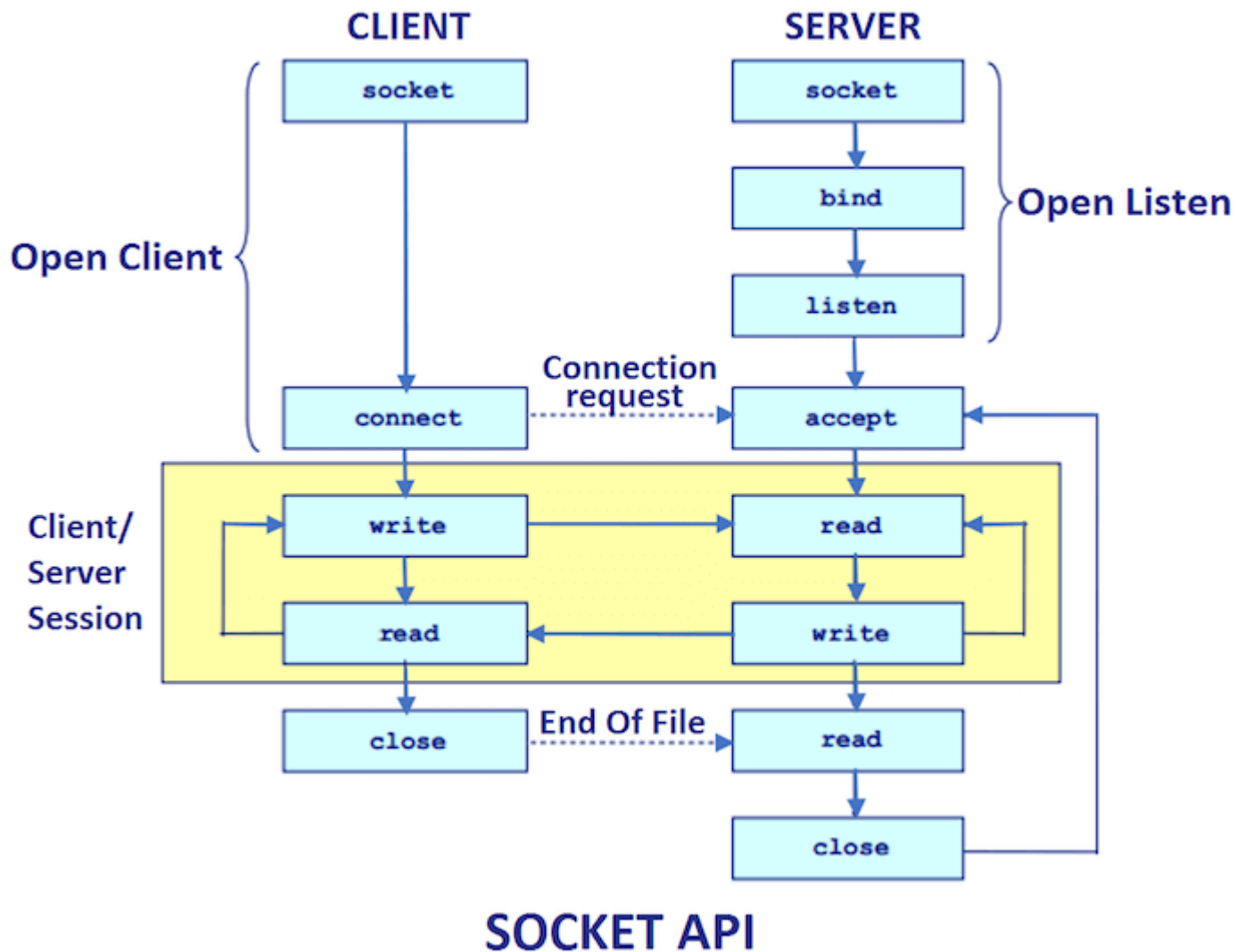
Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

Creating Server:

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You

may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

1. ServerSocket ss=**new** ServerSocket(**6666**);
2. Socket s=ss.accept();//**establishes connection and waits for the client**

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

1. Socket s=**new** Socket("localhost",**6666**);

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

File: MyServer.java

1. **import** java.io.*;
2. **import** java.net.*;
3. **public class** MyServer {
4. **public static void** main(String[] args){
5. **try**{
6. ServerSocket ss=**new** ServerSocket(**6666**);
7. Socket s=ss.accept();//**establishes connection**
8. DataInputStream dis=**new** DataInputStream(s.getInputStream());
9. String str=(String)dis.readUTF();
10. System.out.println("message= "+str);
11. ss.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

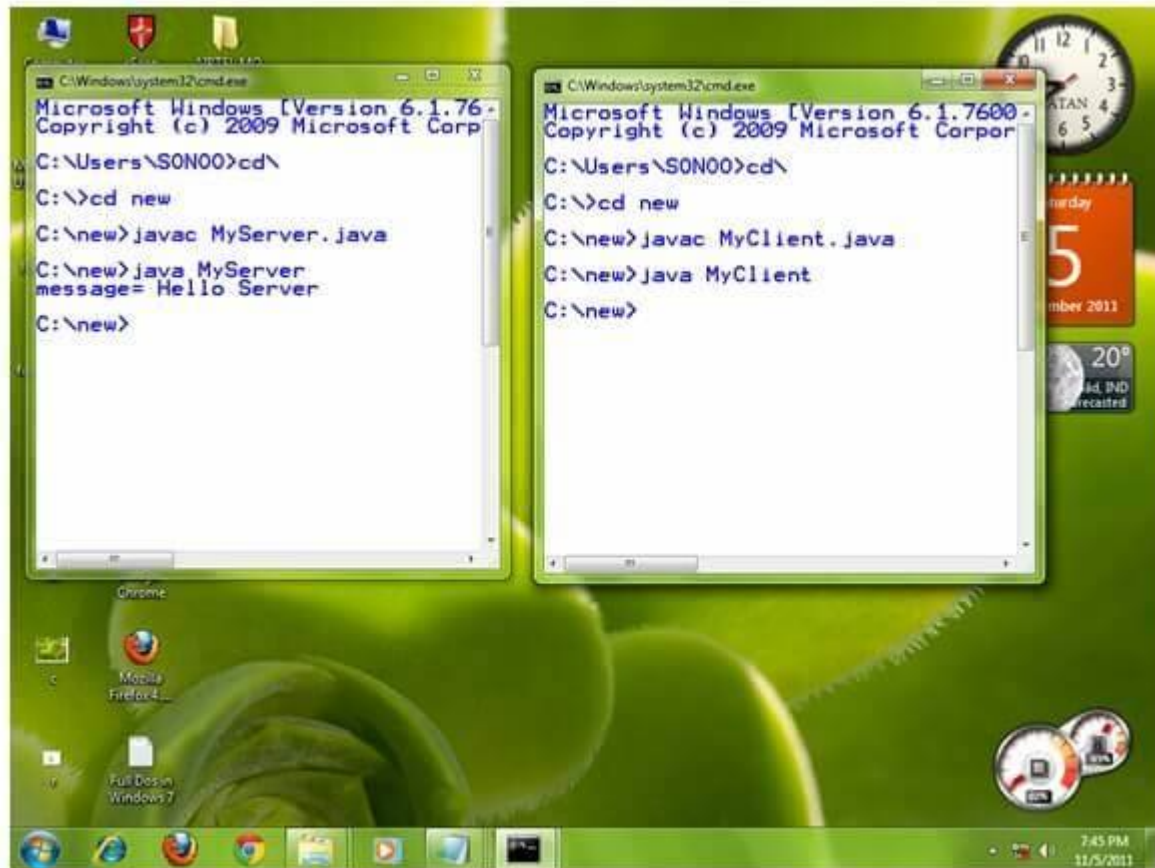
File: MyClient.java

1. **import** java.io.*;
2. **import** java.net.*;
3. **public class** MyClient {
4. **public static void** main(String[] args) {
5. **try**{
6. Socket s=**new** Socket("localhost",**6666**);
7. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
8. dout.writeUTF("Hello Server");
9. dout.flush();
10. dout.close();
11. s.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

[download this example](#)

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



Example of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

File: MyServer.java

1. **import** java.net.*;
2. **import** java.io.*;
3. **class** MyServer{
4. **public static void** main(String args[])**throws** Exception{
5. ServerSocket ss=**new** ServerSocket(3333);
6. Socket s=ss.accept();
7. DataInputStream din=**new** DataInputStream(s.getInputStream());
8. DataOutputStream dout=**new** DataOutputStream(s.getOutputStream());
9. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
- 10.
11. String str="",str2="";
12. **while**(!str.equals("stop")){
13. str=din.readUTF();
14. System.out.println("client says: "+str);
15. str2=br.readLine();
16. dout.writeUTF(str2);
17. dout.flush();
18. }
19. din.close();

```
20. s.close();
21. ss.close();
22. }}
```

File: MyClient.java

```
1. import java.net.*;
2. import java.io.*;
3. class MyClient{
4.     public static void main(String args[])throws Exception{
5.         Socket s=new Socket("localhost",3333);
6.         DataInputStream din=new DataInputStream(s.getInputStream());
7.         DataOutputStream dout=new DataOutputStream(s.getOutputStream());
8.         BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
9.
10.        String str="",str2="";
11.        while(!str.equals("stop")){
12.            str=br.readLine();
13.            dout.writeUTF(str);
14.            dout.flush();
15.            str2=din.readUTF();
16.            System.out.println("Server says: "+str2);
17.        }
18.
19.        dout.close();
20.        s.close();
21.    }}
```

UDP Sockets:

UDP Server-Client implementation in C

There are two major transport layer protocols to communicate between hosts : **TCP** and **UDP**.

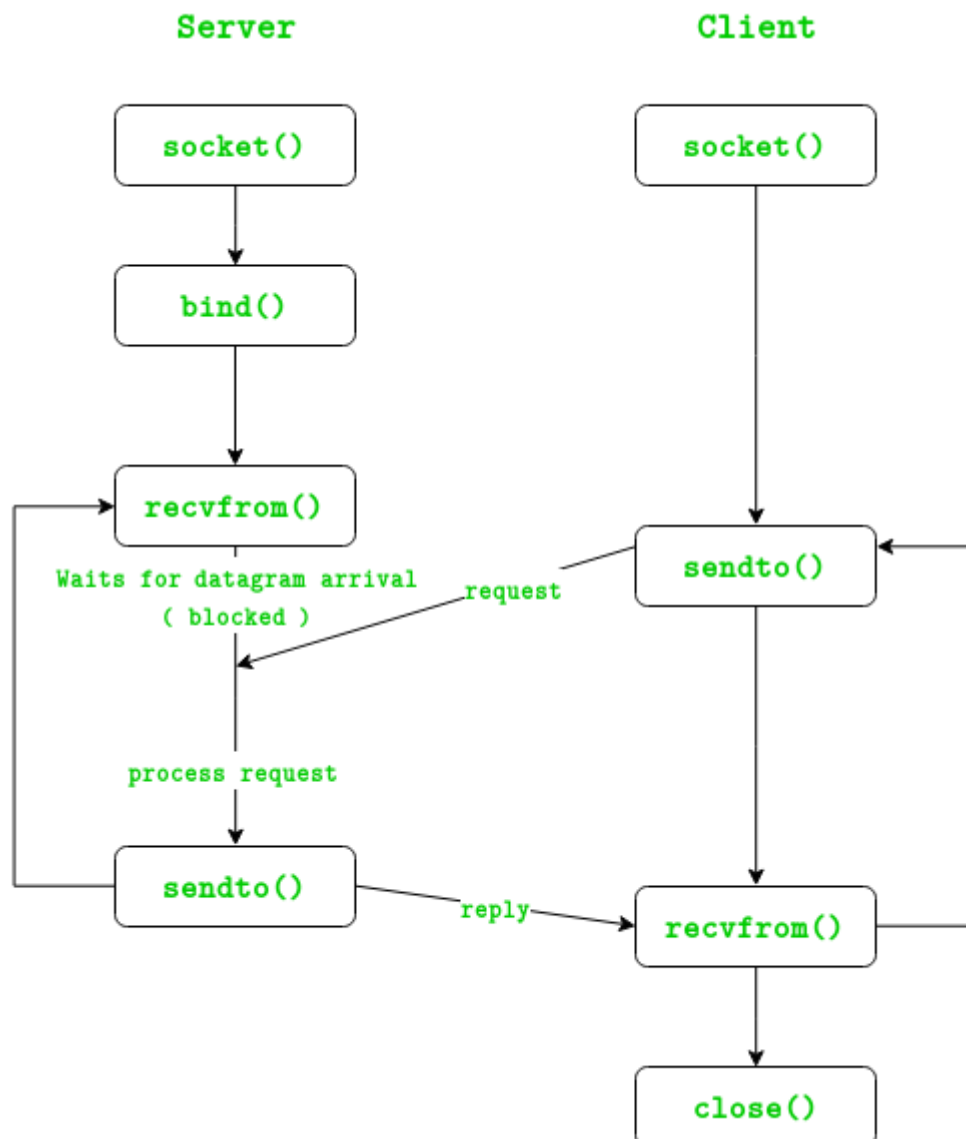
Creating TCP Server/Client was discussed [in a previous post](#).

Prerequisite : [Creating TCP Server/Client](#)

Theory

In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data

to the correct client.



The entire process can be broken down into following steps :

UDP Server :

1. Create UDP socket.
2. Bind the socket to server address.
3. Wait until datagram packet arrives from client.
4. Process the datagram packet and send a reply to client.
5. Go back to Step 3.

UDP Client :

1. Create UDP socket.
2. Send message to server.
3. Wait until response from server is recieved.
4. Process reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

Necessary Functions :

`int socket(int domain, int type, int protocol)`
Creates an unbound socket in the specified domain.
Returns socket file descriptor.

Arguments :**domain** – Specifies the communication

domain (AF_INET for IPv4/ AF_INET6 for IPv6)

type – Type of socket to be created

(SOCK_STREAM for TCP / SOCK_DGRAM for UDP)

protocol – Protocol to be used by socket.

0 means use default protocol for the address family.

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Assigns address to the unbound socket.

Arguments :**sockfd** – File descriptor of socket to be binded**addr** – Structure in which address to be binded to is specified**addrlen** – Size of *addr* structuressize_t sendto(int sockfd, const void *buf, size_t len, int flags,
const struct sockaddr *dest_addr, socklen_t addrlen)

Send a message on the socket

Arguments :**sockfd** – File descriptor of socket**buf** – Application buffer containing the data to be sent**len** – Size of *buf* application buffer**flags** – Bitwise OR of flags to modify socket behaviour**dest_addr** – Structure containing address of destination**addrlen** – Size of *dest_addr* structuressize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
struct sockaddr *src_addr, socklen_t *addrlen)

Receive a message from the socket.

Arguments :**sockfd** – File descriptor of socket**buf** – Application buffer in which to receive data**len** – Size of *buf* application buffer**flags** – Bitwise OR of flags to modify socket behaviour**src_addr** – Structure containing source address is returned**addrlen** – Variable in which size of *src_addr* structure is returned

int close(int fd)

Close a file descriptor

Arguments :**fd** – File descriptor

In the below code, exchange of one hello message between server and client is shown to demonstrate the model.

- **UDPClient.c**

filter_none

```

edit
play_arrow
brightness_4
// Client side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT 8080
#define MAXLINE 1024

// Driver code
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in servaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
        sizeof(servaddr));
    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
        MSG_WAITALL, (struct sockaddr *) &servaddr,
        &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd);

```

```
    return 0;
}
```

Output :

```
$ ./server
Client : Hello from client
Hello message sent.
$ ./client
Hello message sent.
Server : Hello from server
```

Java Beans:

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

Following are the unique characteristics that distinguish a JavaBean from other Java classes –

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "**getter**" and "**setter**" methods for the properties.

JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be **read**, **write**, **read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class –

S.No.	Method & Description
1	getPropertyName() For example, if property name is <i>firstName</i> , your method name would be getFirstName() to read that property. This method is called accessor.
2	setPropertyName() For example, if property name is <i>firstName</i> , your method name would be setFirstName() to write that property. This method is called mutator.

A read-only attribute will have only a **getPropertyName()** method, and a write-only attribute will have only a **setPropertyName()** method.

JavaBeans Example

Consider a student class with few properties –

```
package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable {
    private String firstName = null;
    private String lastName = null;
    private int age = 0;

    public StudentsBean() {
    }
    public String getFirstName(){
        return firstName;
    }
}
```



```

    }
    public String getLastName(){
        return lastName;
    }
    public int getAge(){
        return age;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
    public void setAge(Integer age){
        this.age = age;
    }
}

```

Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows –

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Here values for the scope attribute can be a **page**, **request**, **session** or **application** based on your requirement. The value of the **id** attribute may be any value as long as it is a unique name among other **useBean** declarations in the same JSP.

Following example shows how to use the useBean action –

```

<html>
<head>
    <title>useBean Example</title>
</head>

<body>
    <jsp:useBean id = "date" class = "java.util.Date" />
    <p>The date/time is <%= date %>
</body>
</html>

```

You will receive the following result –

The date/time is Thu Sep 30 11:18:11 GST 2010

Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax –

```

<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
    <jsp:setProperty name = "bean's id" property = "property name"
        value = "value"/>
    <jsp:getProperty name = "bean's id" property = "property name"/>
    .....
</jsp:useBean>

```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax –

```
<html>
<head>
  <title>get and set properties Example</title>
</head>

<body>
  <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">
    <jsp:setProperty name = "students" property = "firstName" value = "Zara"/>
    <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>
    <jsp:setProperty name = "students" property = "age" value = "10"/>
  </jsp:useBean>

  <p>Student First Name:
    <jsp:getProperty name = "students" property = "firstName"/>
  </p>

  <p>Student Last Name:
    <jsp:getProperty name = "students" property = "lastName"/>
  </p>

  <p>Student Age:
    <jsp:getProperty name = "students" property = "age"/>
  </p>

</body>
</html>
```

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed –

Student First Name: Zara

Student Last Name: Ali

Student Age: 10

RMI:

In the previous chapter, we created a sample RMI application where a client invokes a method which displays a GUI window (JavaFX).

In this chapter, we will take an example to see how a client program can retrieve the records of a table in MySQL database residing on the server.

Assume we have a table named **student_data** in the database **details** as shown below.

ID	NAME	BRANCH	PERCENTAGE	EMAIL
1	Ram	IT	85	ram123@gmail.com
2	Rahim	EEE	95	rahim123@gmail.com

Assume the name of the user is **myuser** and its password is **password**.

Creating a Student Class

Create a **Student** class with **setter** and **getter** methods as shown below.

```
public class Student implements java.io.Serializable {
    private int id, percent;
    private String name, branch, email;

    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public String getBranch() {
        return branch;
    }
    public int getPercent() {
        return percent;
    }
    public String getEmail() {
        return email;
    }
    public void setID(int id) {
        this.id = id;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setBranch(String branch) {
        this.branch = branch;
    }
    public void setPercent(int percent) {
        this.percent = percent;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

Defining the Remote Interface

Define the remote interface. Here, we are defining a remote interface named **Hello** with a method named **getStudents ()** in it. This method returns a list which contains the object of the class **Student**.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;
```

```
// Creating Remote interface for our application
```

```
public interface Hello extends Remote {  
    public List<Student> getStudents() throws Exception;  
}
```

Developing the Implementation Class

Create a class and implement the above created **interface**.

Here we are implementing the **getStudents()** method of the **Remote interface**. When you invoke this method, it retrieves the records of a table named **student_data**. Sets these values to the Student class using its setter methods, adds it to a list object and returns that list.

```
import java.sql.*;  
import java.util.*;  
  
// Implementing the remote interface  
public class ImplExample implements Hello {  
  
    // Implementing the interface method  
    public List<Student> getStudents() throws Exception {  
        List<Student> list = new ArrayList<Student>();  
  
        // JDBC driver name and database URL  
        String JDBC_DRIVER = "com.mysql.jdbc.Driver";  
        String DB_URL = "jdbc:mysql://localhost:3306/details";  
  
        // Database credentials  
        String USER = "myuser";  
        String PASS = "password";  
  
        Connection conn = null;  
        Statement stmt = null;  
  
        //Register JDBC driver  
        Class.forName("com.mysql.jdbc.Driver");  
  
        //Open a connection  
        System.out.println("Connecting to a selected database...");  
        conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        System.out.println("Connected database successfully...");  
  
        //Execute a query  
        System.out.println("Creating statement...");  
  
        stmt = conn.createStatement();  
        String sql = "SELECT * FROM student_data";  
        ResultSet rs = stmt.executeQuery(sql);  
  
        //Extract data from result set  
        while(rs.next()) {  
            // Retrieve by column name  
            int id = rs.getInt("id");  
  
            String name = rs.getString("name");
```

```

String branch = rs.getString("branch");

int percent = rs.getInt("percentage");
String email = rs.getString("email");

// Setting the values
Student student = new Student();
student.setID(id);
student.setName(name);
student.setBranch(branch);
student.setPercent(percent);
student.setEmail(email);
list.add(student);
}
rs.close();
return list;
}
}

```

Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMI registry**.

Following is the server program of this application. Here, we will extend the above created class, create a remote object and register it to the RMI registry with the bind name *hello*.

```

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server extends ImplExample {
    public Server() {}
    public static void main(String args[]) {
        try {
            // Instantiating the implementation class
            ImplExample obj = new ImplExample();

            // Exporting the object of implementation class (
            // here we are exporting the remote object to the stub)
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

            // Binding the remote object (stub) in the registry
            Registry registry = LocateRegistry.getRegistry();

            registry.bind("Hello", stub);
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

```

```
}
```

Client Program

Following is the client program of this application. Here, we are fetching the remote object and invoking the method named **getStudents()**. It retrieves the records of the table from the list object and displays them.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.*;

public class Client {
    private Client() {}
    public static void main(String[] args) throws Exception {
        try {
            // Getting the registry
            Registry registry = LocateRegistry.getRegistry(null);

            // Looking up the registry for the remote object
            Hello stub = (Hello) registry.lookup("Hello");

            // Calling the remote method using the obtained object
            List<Student> list = (List) stub.getStudents();
            for (Student s: list) {

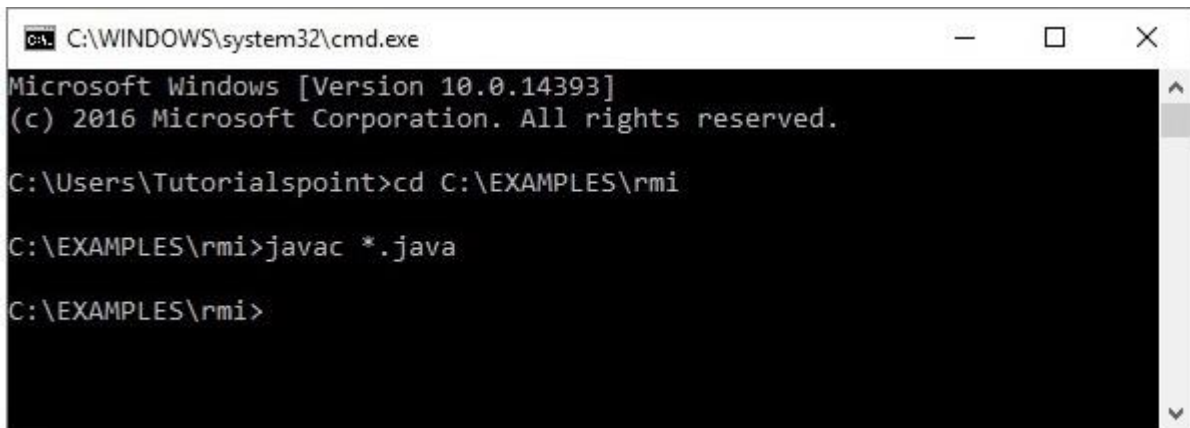
                // System.out.println("bc "+s.getBranch());
                System.out.println("ID: " + s.getId());
                System.out.println("name: " + s.getName());
                System.out.println("branch: " + s.getBranch());
                System.out.println("percent: " + s.getPercent());
                System.out.println("email: " + s.getEmail());
            }
            // System.out.println(list);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Steps to Run the Example

Following are the steps to run our RMI Example.

Step 1 – Open the folder where you have stored all the programs and compile all the Java files as shown below.

Javac *.java



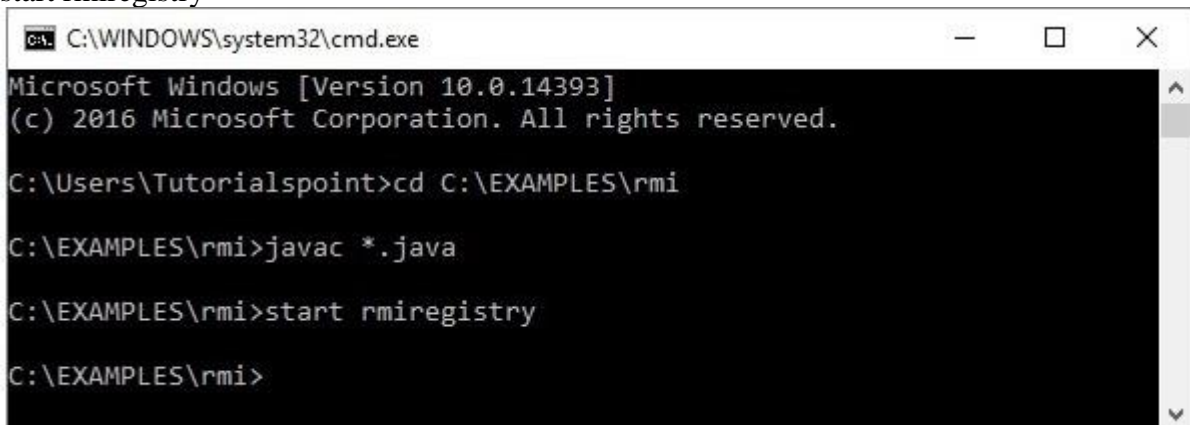
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>
```

Step 2 – Start the **rmi** registry using the following command.
start rmiregistry



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>
```

This will start an **rmi** registry on a separate window as shown below.



```
C:\Program Files\Java\jdk1.8.0_101\bin\rmiregistry.exe
```

Step 3 – Run the server class file as shown below.
Java Server



```
C:\WINDOWS\system32\cmd.exe - java Server

C:\EXAMPLES\rmi>java Server
Server ready
```

Step 4 – Run the client class file as shown below.
java Client

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>java Client
ID: 1
name: Ram
branch: IT
percent: 85
email: ram123@gmail.com
ID: 2
name: Rahim
branch: EEE
percent: 95
email: rahim123@gmail.com
ID: 3
name: Robert
branch: ECE
percent: 90
email: robert123@gmail.com

C:\EXAMPLES\rmi>
```