UNIT 4 ADVANCE JAVA

Structure **Page Nos.**

4.0 INTRODUCTION

This unit will introduce you to the advanced features of Java. To save data, you have used the file system, which gives you functionality of accessing the data but it does not offer any capability for querying on data conveniently.

You are familiar with various databases like Oracle, Sybase, SQL Server etc. They do not only provide the file-processing capabilities, but also organize data in a manner that facilitates applying queries.

Structured Query Language (SQL) is almost universally used in relational database systems to make queries based on certain criteria. In this unit you will learn how you can interact to a database using ***Java Database Connectivity (JDBC)*** feature of Java.

You will also learn about RMI (Remote Method Invocation) feature of Java. This will give the notion of client/server distributed computing. ***RMI*** allows Java objects running on the same or separate computers to communicate with one another via remote method calls.

A request-response model of communication is essential for the highest level of networking. **The Servlets** feature of Java provides functionality to extend the capabilities of servers that host applications accessed via a request-response programming model. In this unit we will learn the basics of Servlet programming.

Java beans are nothing but small reusable pieces of components that you can add to a program without disturbing the existing program code, are also introduced in this unit.

4.1 OBJECTIVES

After going through of this unit you will be able to:

- interact with databases through java programs;
- use the classes and interfaces of the *java.sql* package;

- use basic database queries using Structured Query Language (SQL) in your programs;
- explain Servlet Life Cycle;
- write simple servlets programs;
- explain the model of client/server distributed computing;
- explain architecture of RMI, and
- describe Java Beans and how they facilitate component-oriented software construction.

4.2 JAVA DATABASE CONNECTIVITY

During programming you may need to interact with database to solve your problem. Java provides JDBC to connect to databases and work with it. Using standard library routines, you can open a connection to the database. Basically JDBC allows the integration of SQL calls into a general programming environment by providing library routines, which interface with the database. In particular, Java's JDBC has a rich collection of routines which makes such an interface extremely simple and intuitive.

4.2.1 Establishing A Connection

The first thing to do, of course, is to install Java, JDBC and the DBMS on the working machines. Since you want to interface with a database, you would need a driver for this specific database.

Load the vendor specific driver

This is very important because you have to ensure portability and code reuse. The API should be designed as independent of the version or the vendor of a database as possible. Since different DBMS's have different behaviour, you need to tell the driver manager which DBMS you wish to use, so that it can invoke the correct driver.

For example, an Oracle driver is loaded using the following code snippet:

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

Make the connection

Once the driver is loaded and ready for a connection to be made, you may create an instance of a Connection object using:

```
Connection con = DriverManager.getConnection(url, username, password);
```

Let us see what are these parameters passed to get Connection method of DriverManager class. The first string is the URL for the database including the protocol, the vendor, the driver, and the server the port number. The username and password are the name of the user of database and password is user password.
The connection *con* returned in the last step is an open connection, which will be used to pass SQL statements to the database.

Creating JDBC Statements

A JDBC Statement object is used to send the SQL statements to the DBMS. It is entirely different from the SQL statement. A JDBC Statement object is an open connection, and not any single SQL Statement. You can think of a JDBC Statement object as a channel sitting on a connection, and passing one or more of the SQL statements to the DBMS.

An active connection is needed to create a Statement object. The following code is a snippet, using our Connection object *con*

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

At this point, you will notice that a Statement object exists, but it does not have any SQL statement to pass on to the DBMS.

Creating JDBC PreparedStatement

PreparedStatement object is more convenient and efficient for sending SQL statements to the DBMS. The main feature, which distinguishes PreparedStatement object from objects of Statement class, is that it gives an SQL statement right when it is created. This SQL statement is then sent to the DBMS right away, where it is compiled. Thus, in effect, a PreparedStatement is associated as a channel with a connection and a compiled SQL statement.

Another advantage offered by PreparedStatement object is that if you need to use the same or similar query with different parameters multiple times, the statement can be compiled and optimized by the DBMS just once. While with a normal Statement, each use of the same SQL statement requires a compilation all over again.

PreparedStatements are also created with a Connection method. The following code shows how to create a parameterized SQL statement with three input parameters:

```
  PreparedStatement prepareUpdatePrice
= con.prepareStatement( "UPDATE Employee SET emp_address =? WHERE
   emp_code =“1001” AND emp_name =?");
```

You can see two? symbol in the above PreparedStatement *prepareUpdatePrice*. This means that you have to provide values for two variables emp_address and emp_name in PreparedStatement before you execute it. Calling one of the setXXX methods defined in the class PreparedStatement can provide values. Most often used methods are setInt, setFloat, setDouble, setString, etc. You can set these values before each execution of the prepared statement.

You can write something like:

```
prepareUpdatePrice.setInt(1, 3);
prepareUpdatePrice.setString(2, "Renuka");
prepareUpdatePrice.setString(3, "101, Sector-8,Vasundhara, M.P");
```

Executing CREATE/INSERT/UPDATE Statements of SQL

Executing SQL statements in JDBC varies depending on the intention of the SQL statement. DDL (Data Definition Language) statements such as table creation and table alteration statements, as well as statements to update the table contents, all are executed using the *executeUpdate* method. The following snippet has examples of executeUpdate statements.

```
Statement stmt = con.createStatement();
stmt.executeUpdate("CREATE TABLE Employee " +
"(emp_name VARCHAR2(40), emp_address VARCHAR2(40), emp_sal REAL)" );
stmt.executeUpdate("INSERT INTO Employee " +
  "VALUES ('Archana', '10,Down California', 30000" );
String sqlString = "CREATE TABLE Employee " +
  "(name VARCHAR2(40), address VARCHAR2(80), license INT)" ;
stmt.executeUpdate(sqlString);
```

Since the SQL statement will not quite fit on one line on the page, you can split it into two or more strings concatenated by a plus sign(+).
"INSERT INTO Employee" to separate it in the resulting string from "VALUES".

The point to note here is that the same Statement object is reused rather than to create a new one each time.

When executeUpdate is used to call DDL statements, the return value is always zero, while data modification statement executions will return a value greater than or equal to zero, which is the number of tuples affected in the relation by execution of modification statement.

While working with a PreparedStatement, you should execute such a statement by first plugging in the values of the parameters (as you can see above), and then invoking the executeUpdate on it. For example:

```
int n = prepareUpdateEmployee.executeUpdate() ;
```

Executing SELECT Statements

A query is expected to return a set of tuples as the result, and not change the state of the database. Not surprisingly, there is a corresponding method called execute Query, which returns its results as a ResultSet object. It is a table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The *next method* moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set. A default ResultSet object is not updatable and has a cursor that moves forward only

In the program code given below:

```
 String ename,eaddress;
  float esal;

  ResultSet rs = stmt.executeQuery("SELECT * FROM Employee");
  while ( rs.next() ) {
    ename = rs.getString("emp_name");
    eaddress = rs.getString("emp_address");
    esal = rs.getFloat("emp_salary");
    System.out.println(ename + " address is" + eaddress + " draws salary  " + esal + "
in dollars");
  }
```

The tuples resulting from the query are contained in the variable rs which is an instance of ResultSet. A set is of not much use to you unless you can access each row and the attributes in each row. The?

Now you should note that each invocation of the *next method* causes it to move to the next row, if one exists and returns true, or returns false if there is no remaining row.

You can use the getXXX method of the appropriate type to retrieve the attributes of a row. In the above program code getString and getFloat methods are used to access the column values. One more thing you can observe that the name of the column whose value is desired is provided as a parameter to the method.

Similarly,while working with a PreparedStatement, you can execute a query by first plugging in the values of the parameters, and then invoking the executeQuery on it.

1. ename = rs.getString(1);
   eaddress = rs.getFloat(3);
   esal = rs.getString(2);

2. ResultSet rs = prepareUpdateEmployee.executeQuery() ;

Accessing ResultSet

Now to reach each record of the database, JDBC provides methods like getRow, isFirst, isBeforeFirst, isLast, isAfterLas to access ResultSet.Also there are means to make scroll-able cursors to allow free access of any row in the ResultSet. By default, cursors scroll forward only and are read only. When creating a Statement for a Connection, we can change the type of ResultSet to a more flexible scrolling or updatable model:

```
 Statement stmt = con.createStatement
 ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM Sells");
```

The different options for types are TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE. We can choose whether the cursor is read-only or updatable using the options CONCUR_READ_ONLY, and CONCUR_UPDATABLE.

With the default cursor, we can scroll forward using rs.next(). With scroll-able cursors we have more options:

```
rs.absolute(3);     // moves to the third tuple or row
rs.previous();      // moves back one tuple (tuple 2)
rs.relative(2);     // moves forward two tuples (tuple 4)
rs.relative(-3);    // moves back three tuples (tuple 1)
```

4.2.2 Transactions with Database

When you go to some bank for deposit or withdrawal of money, you get your bank account updated, or in other words you can say some transaction takes place.

JDBC allows SQL statements to be grouped together into a single transaction. Thus, you can ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties using JDBC transactional features.

The Connection object performs transaction control. When a connection is created, by default it is in the *auto-commit* mode. This means that each individual SQL statement is treated as a transaction by itself, and will be committed as soon as its execution is finished.

You can turn off *auto-commit* mode for an active connection with:
con.setAutoCommit(false) ;
And turn it on again if needed with:
con.setAutoCommit(true) ;
Once *auto-commit* is off, no SQL statements will be committed (that is, the database will not be permanently updated) until you have explicitly told it to commit by invoking the commit () method:
con.commit() ;

At any point before commit, you may invoke rollback () to rollback the transaction, and restore values to the last commit point (before the attempted updates).

☞ Check Your Progress 1

1) How is a program written in java to access database?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) What are the different kinds of drivers for JDBC?
……………………………………………………………………………………
……………………………………………………………………………………

3) Write a program code to show how you will perform commit() and rollback().
……………………………………………………………………………………
……………………………………………………………………………………

4) Read the following program assuming mytable already exists in database and answer the questions given below:

   i. What is the use of rs.next()?
   ii. Value of which attribute will be obtained by rs.getString(3).
   iii. If the statement “Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")” is removed from the program what will happen.

```
import java.sql.*;
import java.io.*;
public class TestJDBC
   {
   public static void main(String[] args)
      {
     String dataSourceName = "mp";
      String dbURL = "jdbc:odbc:" + dataSourceName;
      try
      {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection(dbURL, "","");
         Statement s = con.createStatement();
         s.execute("insert into mytable values('AKM',20,'azn')");
         s.executeQuery("select * from mytable ");
         ResultSet rs = s.getResultSet();
            rs.next();
            String n = rs.getString(1);
            System.out.println("Name:"+ n);
      if (rs != null)
      while ( rs.next() )
         {
           System.out.println("Data from column_name: " + rs.getString(1) );
    System.out.println("Data from column_age: " + rs.getInt(2) );
    System.out.println("Data from column_address: " + rs.getString(3) );
      }
   }
   catch (Exception err)
     {
      System.out.println( "Error: " + err );
     }
}
```

4.3 AN OVERVIEW OF RMI APPLICATIONS

Many times you want to communicate between two computers. One of the examples of this type of communication is a chatting program. How do chatting happens or two computers communicate each other? RPC (Remote Procedure Call) is one of the ways to perform this type of communication. In this section you will learn about RMI (Remote Method Invocation).

Java provides RMI (Remote Method Invocation), which is *"a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism."*

RPC (Remote Procedure Call) organizes the types of messages which an application can receive in the form of functions. Basically it is a management of streams of data transmission.

RMI applications often comprised two separate programs: *a server and a client*.
A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects.
A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

There are three processes that participate in developing applications based on remote method invocation.

1. The *Client* is the process that is invoking a method on a remote object.
2. The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

4.3.1 Remote Classes and Interfaces

A *Remote* class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:

1. Within the address space where the object was constructed, the object is an ordinary object, which can be used like any other object.
2. Within other address spaces, the object can be referenced using an object handle While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

For simplicity, an instance of a Remote class is called a *remote object*.

A Remote class has two parts: the interface and the class itself.

The Remote interface must have the following properties:

Interface must be public.
Interface must extend the *java.rmi.Remote* interface. Every method in the interface must declare that it throws java.rmi.RemoteException. Maybe other exceptions also ought to be thrown.

The Remote class itself has the following properties:

It must implement a Remote interface.

It should extend the *java.rmi.server.UnicastRemoteObject* class. Objects of such a class exist in the address space of the server and can be invoked remotely. While there are other ways to define a Remote class, this is the simplest way to ensure that objects of a class can be used as remote objects.

It can have methods that are not in its Remote interface. These can only be invoked locally. It is not necessary for both the Client and the Server to have access to the definition of the Remote class.

The Server requires the definition of both the Remote class and the Remote interface, but the client only uses the Remote interface.

All of the Remote interfaces and classes should be compiled using *javac*. Once this has been completed, the stubs and skeletons for the Remote interfaces should be compiled by using the *rmic stub* compiler. The stub and skeleton of the example Remote interface are compiled with the command:
rmic <filename.class>

4.3.2 RMI Architecture

It consists of three layers as given in Figure 1

1. Stub/Skeleton layer – client-side stubs and server-side skeletons.
2. Remote reference layer-invocation to single or replicated object
3. Transport layer-connection set up and management, also remote object tracking.

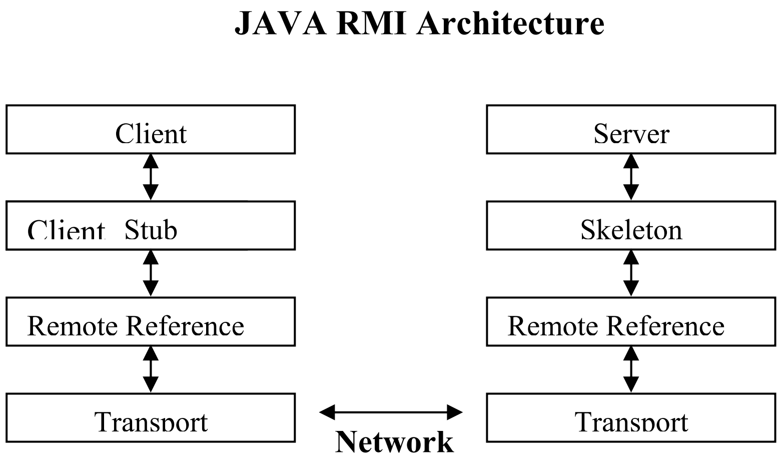


Figure 1: Java RMI Architecture

4.3.3 RMI Object Hierarchy

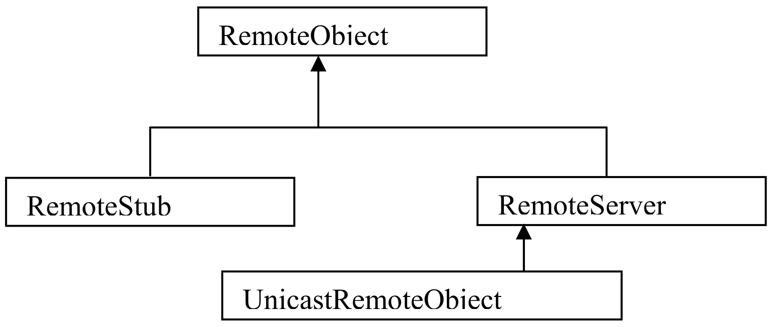


Figure 2:RemoteObject Hierarchy

In *Figure 2* above, the RMI Object Hierarchy is shown. The most general feature set associated with RMI is found in the java.rmi.Remote interface. Abstract class java.rmi.server.RemoteObject supports the needed modifications to the Java object model to cope with the indirect references.

Remote Server is a base class, which encapsulates transport semantics for RemoteObjects. Currently RMI ships with a UnicastRemoteObject(single object)

A server in RMI is a named service which is registered with the RMI registry, and listens for remote requests. For security reasons, an application can bind or unbind only in the registry running on the same host.

4.3.4 Security

One of the most common problems with RMI is a failure due to security constraints. Let us see Java the security model related to RMI. A Java program may specify a security manager that determines its security policy. A program will not have any security manager unless one is specified. You can set the security policy by constructing a *SecurityManager object* and calling the *setSecurityManager* method of the *System* class. Certain operations require that there be a security manager. For example, RMI will download a Serializable class from another machine only if there is a security manager and the security manager permits the downloading of the class from that machine. The RMISecurityManager class defines an example of a security manager that normally permits such download. However, many Java installations have instituted security policies that are more restrictive than the default. There are good reasons for instituting such policies, and you should not override them carelessly.

Creating Distributed Applications Using RMI

The following are the basic steps be followed to develop a distributed application using RMI:

- Design and implement the components of your distributed application.
- Compile sources and generate stubs.
- Make classes network accessible.
- Start the application.

Compile Sources and Generate Stubs

This is a two-step process. In the first step you use the javac compiler to compile the source files, which contain the implementation of the remote interfaces and implementations, of the server classes and the client classes. In the second step you use the rmic compiler to create stubs for the remote objects. RMI uses a remote object's stub class as a proxy in clients so that clients can communicate with a particular remote object.

Make Classes Network Accessible

In this step you have to make everything: the class files associated with the remote interfaces, stubs, and other classes that need to be downloaded to clients, accessible via a Web server.

Start the Application

Starting the application includes running the RMI remote object registry, the server, and the client.

☞ Check Your Progress 2

1) What is Stub in RMI?

……………………………………………………………………………………

……………………………………………………………………………………

2) What are the basic actions performed by receiver object on server side?

……………………………………………………………………………………

……………………………………………………………………………………

3) What is the need of and Registry Service of RMI?

……………………………………………………………………………………

……………………………………………………………………………………

4.4 JAVA SERVLETS

In this section you will be introduced to server side-programming. Java has utility known as servlets for server side-programming.

A *servlet* is a class of Java programming language used to extend the capabilities of servers that host applications accessed via a request-response programming model.

Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. Java Servlet technology also defines HTTP-specific servlet classes. The javax.servlet and java.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.

When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as do get and do Post, for handling HTTP-specific services.

In this section we will focus on writing servlets that generate responses to HTTP requests. Here it is assumed that you are familiar with HTTP protocol.

4.4.1 Servlet Life Cycle

The container in which the servlet has been deployed controls the life cycle of a servlet. When a request is mapped to a servlet, the container performs the following steps.

Loads the servlet class.
Creates an instance of the servlet class.
Initializes the servlet instance by calling the init() method.
When servlet is executed it invokes the service method, passing a request and response object.

If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's destroy method.

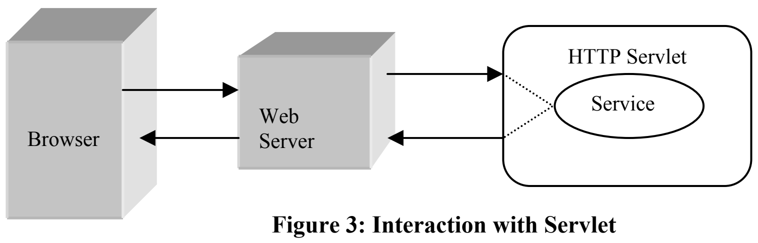


Figure 3: Interaction with Servlet

Servlets are programs that run on servers, such as a web server. You all do net surfing and well known the data on which the web is submitted and you get the respond accordingly. On web pages the data is retrieved from the corporate databases, which should be secure. For these kinds of operations you can use servlets.

4.4.2 GET and POST Methods

The GET methods is a request made by browsers when the user types in a URL on the address line, follows a link from a Web page, or makes an HTML form that does not specify a METHOD. Servlets can also very easily handle POST requests, which are generated when someone creates an HTML form that specifies METHOD="POST".

The program code given below will give you some idea to write a servlet program:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SomeServlet extends HttpServlet
    {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
  // Use "request" to read incoming HTTP headers (e.g. cookies)
  // and HTML form data (e.g. data the user entered and submitted)

  // Use "response" to specify the HTTP response line and headers
  // (e.g. specifying the content type, setting cookies).
 PrintWriter out = response.getWriter();
  // Use "out" to send content to browser
 }
}
```

To act as a servlet, a class should extend HttpServlet and override doGet or doPost (or both), depending on whether the data is being sent by GET or by POST. These methods take two arguments: an HttpServletRequest and an HttpServletResponse objects.
The HttpServletRequest has methods for information about incoming information such as FORM data, HTTP request headers etc.
The httpServletResponse has methods that let you specify the HTTP response line (200, 404, etc.), response headers (Content-Type, Set-Cookie, etc.), and, most importantly, a PrintWriter used to send output back to the client.

A Simple Servlet: Generating Plain Text

Here is a simple servlet that just generates plain text:

```
//Program file name: HelloWorld.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet
 {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
   PrintWriter out = response.getWriter();
   out.println("Hello World");
    }
 }
```

Compiling and Installing the Servlet

Note that the specific details for installing servlets vary from Web server to Web server. Please refer to the Web server documentation for definitive directions. The on-line examples are running on Java Web Server (JWS) 2.0, where servlets are expected to be in a directory called servlets in the JWS installation hierarchy.

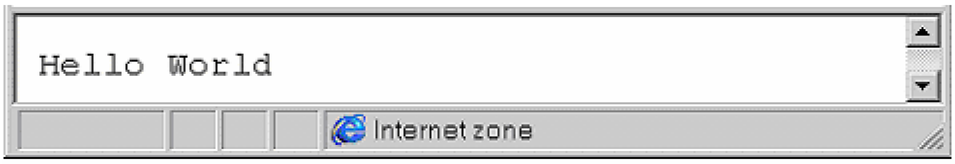
You have to set the CLASSPATH to point to the directory above the one actually containing the servlets. You can then compile normally from within the directory.

```
DOS> set CLASSPATH=C:\JavaWebServer\servlets;%CLASSPATH%
DOS> cd C:\JavaWebServer\servlets\
DOS> javac HelloWorld.java
```

Running the Servlet

With the Java Web Server, servlets are placed in the servlets directory within the main JWS installation directory, and are invoked via *http://host/servlet/ServletName*. Note that the directory is servlets, plural, while the URL refers to servlet, singular. Other Web servers may have slightly different conventions on where to install servlets and how to invoke them. Most servers also let you define aliases for servlets, so that a servlet can be invoked via *http://host/any-path/any-file.html*.

The Url that you will give on the explorer will be:
http://localhost:8080/servlet/HelloWorld then you will get the output as follows:



```
Hello World
```

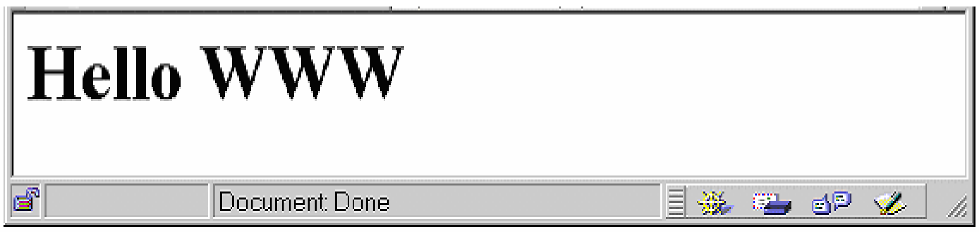
A Servlet that Generates HTML

Most servlets generate HTML, not plain text as in the previous example. To do that, you need two additional steps: tell the browser that you're sending back HTML, and modify the println statements to build a legal Web page. First set the Content-Type response header. In general, headers can be set via the setHeader method of HttpServletResponse, but setting the content type is such a common task that there is also a special setContentType method just for this purpose. You need to set response headers *before* actually returning any of the content via the PrintWriter.
Here is an example for the same:

```
//HelloWWW.java

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWWW extends HttpServlet
{
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
                              "Transitional//EN\">\n" +
              "<HTML>\n" +
            "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
            "<BODY>\n" +
            "<H1>Hello WWW</H1>\n" +
             "</BODY></HTML>");
  }
}
```

URL: http://localhost:8080/servlet/HelloWWW
HelloWWW Output:



Hello WWW

Document: Done

4.4.3 Session Handling

It is essential to track client's requests. To perform this task, Java servlets offers two different ways:

1. It is possible to save information about client state on the server using a *Session* object
2. It is possible to save information on the client system using cookies.

*A **session** is sequence of HTTP requests, from the same client, over a period of time.*

HTTP is a **stateless protocol.** If a client makes a **series** of requests on a server, HTTP provides no help whatsoever to determine if those requests originated from the **same** client. There is no way in HTTP to link two separate requests to the same client. Hence, there is no way to maintain state between client requests in HTTP.

You need to maintain the state on the web for e-commerce type of applications. Just like other software systems, web applications want and need state. The classic web application example is the shopping cart that maintains a list of items you wish to purchase at a web site. The shopping cart's state is the items in the shopping basket at any given time. This state, or shopping items, needs to be maintained over a series of client requests. HTTP alone cannot do this; it needs help.

Now the question arises, for how long can you maintain the state of the same client? Of course, this figure is application-dependent and brings into play the concept of a web session. If a session is configured to last for 30 minutes, once it has expired the client will need to start a new session. Each session requires a unique identifier that can be used by the client.

There are various ways through which you maintain the state.

- Hidden Form Fields
- URL Rewriting
- Session Handling
- Cookies.

Hidden Form Fields

Hidden form fields are HTTP tags that are used to store information that is invisible to the user. In terms of session tracking, the hidden form field would be used to hold a client's unique session id that is passed from the client to the server on each HTTP request. This way the server can extract the session id from the submitted form, like it does for any of form field, and use it to identify which client has made the request and act accordingly.

For example, using servlets you could submit the following search form:

```
<form method="post" action="/servlet/search">
  <input type="text" name="searchtext">
  <input type="hidden" name="sessionid" value="1211xyz">
  ...
</form>
```

When it is submitted to the servlet registered with the name ***search***, it pulls out the sessionid from the form as follows:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
{
  ...
  String theSessionId = request.getParameterValue("sessionid");
  if( isAllowToPerformSearch(theSessionId) )
  {

  }
  ...

 }
```

In this approach' the search servlet gets the session id from the hidden form field and uses it to determine whether it allows performing any more searches.

Hidden form fields implement the required ***anonymous*** session tracking features the client needs but not without cost. For hidden fields to work the client must send a hidden form field to the server and the server must always return that same hidden form field. This tightly coupled dependency between client requests and server responses requires sessions involving hidden form fields to be an unbreakable chain of dynamically generated web pages. If at any point during the session the client accesses a static page that is not point of the chain, the hidden form field is lost, and with it the session is also lost.

URL Rewriting

URL rewriting stores session details as part of the URL itself. You can see below how we look at request information for our search servlet:

i) http://www.archana.com/servlet/search
ii) http://www.archana.com/servlet/search/23434abc
iii) http://www.archana.com/servlet/search?sessionid=23434abc

For the original servlet [i] the URL is clean. In [ii] we have URL re-written at the server to add extra path information as embedded links in the pages we send back to the client. When the client clicks on one of these links, the search servlet will do the following:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
{
  ...
  String sessionid = request.getPathInfo(); // return 2343abc from [ii]
  ...
}
```

Extra path information work for both GET and POST methods involved from inside as well as outside of forms with static links.

Technique [iii] simply re-writes the URL with parameter information that can be accessed as follows:
request.getParameterValue("sessionid");

URL re-writing, like, hidden forms provide a means to implement anonymous session tracking. However, with URL rewriting you are not limited to forms and you can re-write URLs in static documents to contain the required session information. But URL re-writing suffers from the same major disadvantage that hidden form fields do, in that they must be dynamically generated and the chain of HTML page generation cannot be broken.

Session object

A *HttpSession* object (derived from *Session* object) allows the servlet to solve part of the HTTP stateless protocol problems. After its creation a *Session* is available until an explicit invalidating command is called on it (or when a default timeout occurs). The same *Session* object can be shared by two or more cooperating servlets. This means each servlet can track client's service request history.

A servlet accesses a *Session* using the *getSession()* method implemented in the *HttpServletRequest* interface.
getSession() method returns the *Session* related to the current *HttpServletRequest*.

Note:

1. It is important to remember that each instance of *HttpServletRequest* has its own *Session*. If a *Session* object has not been created before, *getSession()* creates a new one.
2. *HttpSession* interface implements necessary methods to manage with a session.

Following are the various methods related to session tracking:

public abstract String getId( ): Returns a string containing session's name. This name is unique and is set by *HttpSessionContext()*.

public abstract void putValue (String Name, Object Value): Connect the object *Value* to the Session object identified by *Name* parameter. If another object has been connected to the same session before, it is automatically replaced.

public abstract Object getValue (String name): Returns the object currently held by current session. It returns a *null* value if no object has been connected before to the session.

public abstract void removeValue (String name): Remove, if existing, the object connected to the session identified by the *Name* parameter.

public abstract void invalidate( ): Invalidate the session.

Cookies

A cookie is a text file with a name, a value and a set of attributes.

Using JSDK (Java Servlet Development Kit) it is possible to save client's state sending cookies. Cookies are sent from the server and saved on client's system. On client's system cookies are collected and managed by the web browser. When a cookie is sent, the server can retrieve it in a successive client's connection. Using this strategy it is possible to track client's connections history.

Cookie class of Java is derives directly from the *Object* class. Each Cookie object instance has some attributes like max age, version, server identification, comment.

Below are some cookie methods:

public Cookie (String Name, String Value): Cookie class' constructor. It has two parameters. The first one is the name that will identity the cookie in the future; the second one, *Value*, is a text representing the cookie value. Notice that *Name* parameter must be a "token" according to the standard defined in RFC2068 and RFC2109.

Public String getName( ): Returns cookie's name. A cookie name is set when the cookie is created and can't be changed.

public void setValue(String NewValue): This method can be used to set or change cookie's value.

public String getValue( ): Returns a string containing cookie's value.

public void setComment(Sting Comment): It is used to set cookie's comment attribute.

public String getComment( ): Returns cookie's comment attribute as a string.

public void setMaxAge (int MaxAge): Sets cookie's max age in seconds. This means client's browser will delete the cookie in *MaxAge* seconds. A negative value indicate the cookie has to be deleted when client's web browser exits.

public int getMaxAge( ): Returns cookie's max age.

☞ Check Your Progress 3

1) What are the advantages of Servlets?

…………………………………………………………………………………………

…………………………………………………………………………………………

2) What is session tracking?

…………………………………………………………………………………………

…………………………………………………………………………………………

3) What is the difference between doGet() and doPost()?

…………………………………………………………………………………………

…………………………………………………………………………………………

4) How does HTTP Servlet handle client requests?

…………………………………………………………………………………………

…………………………………………………………………………………………

4.5 JAVA BEANS

Java Beans are reusable software component model which allow a great flexibility and addition of features in the existing piece of software. You will find it very interesting and useful to use them by linking together the components to create applets or even new beans for reuse by others. Graphical programming and design environments often called builder tools give a good visual support to bean programmers. The builder tool does all the work of associating of various components together.

A "JavaBeans-enabled" builder tool examines the Bean's patterns, discern its features, and exposes those features for visual manipulation. A builder tool maintains Beans in a palette or toolbox. You can select a Bean from the toolbox, drop it into a form, modify its appearance and behaviour, define its interaction with other Beans, and compose it into applets, application, or new Bean. All this can be done without writing a line of code.

Definition: *A Java Bean is a reusable software component that can be visually manipulated in builder tools*

To understand the precise meaning of this definition of a Bean, you must understand the following terms:

- Software component
- Builder tool
- Visual manipulation.

Reusable software components are designed to apply the power and benefit of reusable, interchangeable parts from other industries to the field of software construction. Other industries have long profited from reusable components. Reusable electronic components are found on circuit boards. A typical part in your car can be replaced by a component made from one of many different competing manufacturers. Lucrative industries are built around parts construction and supply in most competitive fields. The idea is that standard interfaces allow for interchangeable, reusable components.

Reusable software components can be simple like familiar push buttons, text fields list boxes, scrollbars, dialogs boxes etc.

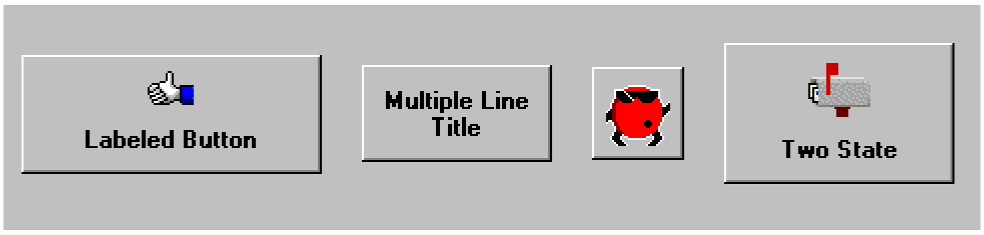


Figure 4: Button Beans

Features of JavaBeans

- Individual Java Beans will vary in functionality, but most share certain common defining features.
- Support for **introspection** allowing a builder tool to analyze how a bean works.
- Support for **customization** allowing a user to alter the appearance and behaviour of a bean.
- Support for **events** allowing beans to fire events, and informing builder tools about both the events they can fire and the events they can handle.
- Support for **properties** allowing beans to be manipulated programmatically, as well as to support the customization mentioned above.
- Support for **persistence** allowing beans that have been customized in an application builder to have their state saved and restored. Typically persistence is used with an application builder's save and load menu commands to restore any work that has gone into constructing an application.

It is not essential that Beans can only be primarily with builder tools. Beans can also be manually manipulated by programmatic interfaces of Text tools. All key APIs, including support for events, properties, and persistence, have been designed to be easily read and understood by human programmers as well as by builder tools.

BeanBox

BeanBox is a utility from the JavaBeans Development Kit (*BDK).* Basically the BeanBox is a test container for your JavaBeans. It is designed to allow programmers to preview how a Bean created by user will be displayed and manipulated in a builder tool. The BeanBox is not a builder tool.It allows programmers to preview how a bean will be displayed and used by a builder tool.

Example of Java Bean Class

Create a new SimpleBean.java program containing the code given below:

/WEB-INF/classes/com/myBean/bean/test/ folder

```
package com.mybean.bean.test;
public class SimpleBean implements java.io.Serializable
{
        /* Properties */
        private String ename = null;
        private int eage = 0;
        /* Empty Constructor */
        public SimpleBean() {}
        /* Getter and Setter Methods */
        public String getEname()
       {
                return ename;
        }
        public void setEname(String s)
       {
                ename = s;
        }
        public int getAge()
       {
                return eage;
        }

        public void setAge(int i)
        {
                eage = i;
        }
}
```

The class SimpleBean implements java.io.Serializable interface.

There are two variables which hold the name and age of a employee. These variables inside a JavaBean are called properties. These properties are private and are thus not directly accessible by other classes. To make them accessible, methods are defined.

Compiling JavaBean

You can compile JavaBean like you compile any other Java Class file. After compilation, a SimpleBean.class file is created and is ready for use.
Finally you can say, JavaBeans are Java classes which adhere to an extremely simple coding convention. All you have to do is to implement java.io.Serializable interface,

use a public empty argument constructor and provide public methods to get and set the values of private variables (properties).

☞ Check Your Progress 4

1) What are JavaBeans?
……………………………………………………………………………………
……………………………………………………………………………………

2) What do you understand by Introspection?
……………………………………………………………………………………
……………………………………………………………………………………

3) What is the difference between a JavaBean and an instance of a normal Java class?
……………………………………………………………………………………
……………………………………………………………………………………

4) In a single line answer what is the main responsibility of a Bean Developer.
……………………………………………………………………………………
……………………………………………………………………………………

4.6 SUMMARY

In this unit you have learn Java JDBC are used to connect databases and work with it. JDBC allows the integration of SQL call into a general programming environment. Vender specific drivers are needed in JDBC programming to make a code portable. getConnection() method of DriverManager class is used to create connection object. By PreparedStatement similar queries can be performed in efficient way. Tuples in ResultSet are accessed by using next () method.

Distributed programming can be done using Java RMI (Remote Methods Invocation). Every RMI program has two sets one for client side and other for server side. Remote interface is essentially implemented in RMI programs.

Servlets are used with web servers. The HttpServlet class is an extension of GenericServlet that include methods for handling HTTP. HTTP request for specific data are handled by using doGet () and doPost () methods of HttpServlet. HttpSession objects are used to solve the problems of HTTP caused due to the stateless nature of HTTP.

Java Beans are a new dimension in software component model. Beans provide introspection and persistency.

4.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Programs are written according to the JDBC driver API would talk to the JDBC driver Manager. JDBC driver Manager would use the drivers that were plugged into it at that moment to access the actual database.

2) Four types of drivers are there for JDBC. They are:
   a) JDBC-ODBC Bridge Driver: Talks to an ODBC connection using largely non-Java code.

b) Native API,(Partly Java) : Uses foreign functions to talk to a non-Java API; the non-Java component talks to the database any way it likes.(Written partly in Java and partly in native code, that communicate with the native API of a database.

c) Net Protocol (pure Java): Talks to a middleware layer over a network connection using the middleware’s own protocol (Client library is independent of the actual database).

d) Native Protocol (pure Java): Talks directly to the RDBMS over a network connection using an RDBMS-specific protocol. (pure Java library that translates JDBC requests directly to a database-specific protocol.

3) //It is assumed that Employee database is already existing.
```
con.setAutoCommit(false);
The Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO Employee VALUES('Archana', 'xyz',
30000)" );
con.rollback();
stmt.executeUpdate("INSERT INTO Sells Employee('Archie', 'ABC', 40000)" );
con.commit();
con.setAutoCommit(true);
```

4)

i. rs.next() is used to move to next row of the table.

ii. rs.getString(3) will give the value of the third attribute in the current row. In this program the third attribute is Address.

iii The statement “Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")” is essential for the program execution. Database driver cannot be included in JDBC runtime method. If you want to remove this statement from program you must have to provide jdbc.divers property using command line parameter.

Check Your Progress 2

1) When you invoke a remote method on a remote object’ the remote method calls a method of java programming language that is encapsulated in a surrogate object called Stub. The Following information is built by Stub:

i. An identifier of the remote object to be used.
ii. A description of the method to be called.
iii. The marshalled parameters.

2) The basic actions performed by receiver object on server side are:

i. Unmarshaling of the parameters.
ii. Locating the object to be called.
iii. Calling the desired method
iv. Capturing the marshals and returning the value or exception of the call.
v. Sending a package consisting of the marshalled return data back to the stub on the client.

3) RMI Registry is required to provide RMI Naming Service which is used to simplify the location of remote objects. The naming service is a JDK utility called rmiregistry that runs at a well-known address.

Check Your Progress 3

1) Java servlets are more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI than many alternative CGI-like technologies.

   Following are the advantages of Services

   Efficient. With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous requests to the same CGI program, then the code for the CGI program is loaded into memory N times. With servlets, however, there are *N* threads but only a single copy of the servlet class.

   Convenient. Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.

   Powerful. Java servlets let us easily do several things that are difficult or impossible with regular CGI. For example servlets can talk directly to the Web server (regular CGI programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each other, making useful things like database connection pools easy to implement.

   Portable. Servlets are written in Java and follow a well-standardized API. Consequently, servlets written for, say I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft IIS, or WebStar. Servlets are supported directly or via a plugin on almost every major Web server.

2) Session tracking is a concept which allows you to maintain a relationship between two successive requests made to a server on the Internet by the same client. Servlet's Provide an API named HttpSession is used in session tracking programming.

3) i. The doGet() method is limited with 2k of data only to be sent, but this limitation is not with doPost() method.
   ii. A request string for doGet() looks like the following:

   http://www.abc.com/svt1?p1=v1&p2=v2&...&pN=vN
   But doPost() method does not need a long text tail after a servlet name in a request.

4) An HTTP Servlet handles client requests through its service method, which supports standard HTTP client requests. The service method dispatches each request to a method designed to handle that request.

Check Your Progress 4

1) Java Beans are components that can be used to assemble a larger Java application. Beans are basically classes that have properties, and can trigger events. To define a property, a bean writer provides accessor methods which are used to get and set the value of a property.

2) Introspection is the process of implicitly or explicitly interrogating Bean.
   Implicit Introspection: Bean runtime supplies the default introspection mechanism which uses the Reflection API and a well established set of Naming Conventions.

Explicit Introspection: A bean designer can provide additional information through an object which implements the Bean Info interface.

In a nutshell, Introspection is a how a builder or designer can get information about how to connect a Bean with an Application.

3) The difference in Beans from typical Java classes is ***introspection***. Tools that recognize predefined patterns in method signatures and class definitions can "look inside" a Bean to determine its properties and behavior. A Bean's state can be manipulated at the time it is being assembled as a part within a larger application. The application assembly is referred to as ***design time*** in contrast to ***run time***. In order for this scheme to work, method signatures within Beans must follow a certain pattern for introspection tools to recognize how Beans can be manipulated, both at design time, and run time.

4) To minimize the effort in turning a component into a Bean.