UNIT I: AN OVERVIEW OF SERVLETS, JSP TECHNOLOGY AND SERVLET BASICS

A Servlets jobs, Why build web pages dynamically?, Advantages of Servlets over traditional CGI, The Role of JSP, Installing & Configuring the JDK & Apache Tomcat, Testing your setup, Web application – A Preview, Basic Servlet structure, A Servlet that generate plain text, A Servlet that generate HTML text, A Servlet package, The Servlet life cycle, The Single Thread model interface, Servlet debugging

A SERVLETS JOBS

Servlets are Java programs that run on Web or application servers, acting as a middle layer between requests coming from Web browsers or other HTTP clients and databases or applications on the HTTP server.

Their job is to perform the following tasks, as illustrated in Figure:

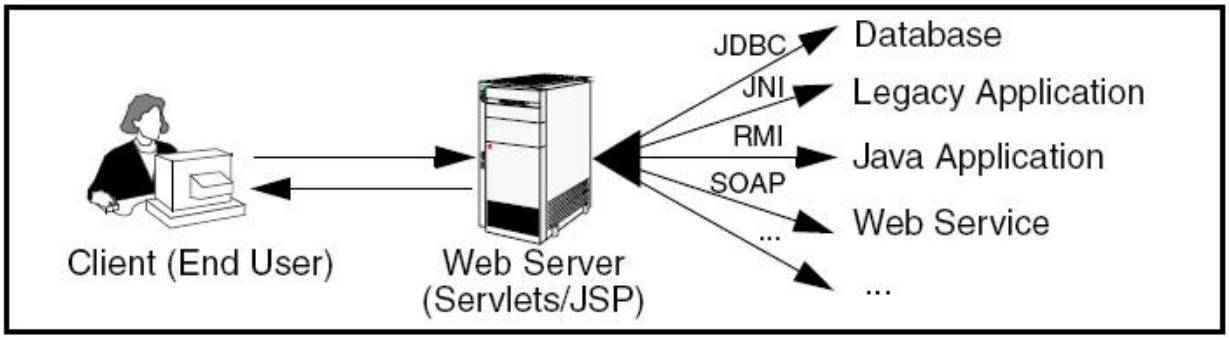


Fig: Servlet Job

1. **Read the explicit data sent by the client.**

   The end user normally enters this data in an HTML form on a Web page. However, the data could also come from an applet or a custom HTTP client program.

2. **Read the implicit HTTP request data sent by the browser.**
   Above Figure shows a single arrow going from the client to the Web server, but there are really *two* varieties of data: the explicit data that the end user enters in a form and the behind-the-scenes HTTP information.
   The HTTP information includes cookies, information about media types and compression schemes the browser understands, etc.
3. **Generate the results.**
   This process may require talking to a database (**JDBC**-Java Database Connectivity), executing an **RMI** (Remote Method Invocation)Application or executing **EJB** (Enterprise Java Beans) call, invoking a **Web service** or Invoking **JNI** (Java Native

Interface) Legacy Application (Application Written in C/C++) , or computing the response directly, and then creates a HTML page as per the user requirement .

4. **Send the explicit data (i.e., the document) to the client.**
   This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), or even a compressed format like gzip. But, HTML is the most common format, so an important servlet task is to wrap the results inside of HTML.

5. **Send the implicit HTTP response data.**
   Above Figure shows a single arrow going from the servlet to the client. But, there are really *two* types of data sent: the document itself and the behind-the-scenes HTTP information. Both varieties are important for effective development.

   Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML, PDF, DOC, XLS etc.), setting cookies and caching parameters, and other such tasks.

WHY BUILD WEB PAGES DYNAMICALLY?

In many cases, a static result is not sufficient, and a page needs to be generated for each request as per the user data.

There are a number of reasons why Web pages need to be built on-the-fly:

- **The Web page is based on data sent by the client.**
  For example, the results page from search engines and order confirmation pages at online stores are specific to particular user requests.

  You don't know what to display until you read the data that the user submits. Just remember that the user submits two kinds of data: explicit (i.e., HTML form data) and implicit (i.e., HTTP request headers). Either kind of input can be used to build the output page. In particular, it is quite common to build a user-specific page.

- **The Web page is derived from data that changes frequently.**

  If the page changes for every request, then you surely need to build the response at request time.
  If it changes only periodically, however, you could do it two ways: you could periodically build a new Web page on the server (independently of client requests), or you could wait and only build the page when the user requests it.

  The right approach depends on the situation, but sometimes it is more convenient to do the latter: wait for the user request. For example, a weather report or news headlines site might build the pages dynamically, perhaps returning a previously built page if that page is still up to date.

- **The Web page uses information from corporate databases or other server-side sources.**

  If the information is in a database, you need server-side processing even if the client is using dynamic Web content such as an applet.

THE ADVANTAGES OF SERVLETS OVER "TRADITIONAL" CGI:

Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI(Common Gateway Interface) and many alternative CGI-like technologies.

Efficient

With traditional CGI, a new process is started for each HTTP request. If the CGI program itself is relatively short, the overhead of starting the process can dominate the execution time. With servlets, the Java virtual machine stays running and handles each request with a lightweight Java thread, not a heavyweight operating system process.

Similarly, in traditional CGI, if there are *N* requests to the same CGI program, the code for the CGI program is loaded into memory *N* times. With servlets, however, there would be *N* threads, but only a single copy of the servlet class would be loaded. This approach reduces server memory requirements and saves time by instantiating fewer objects.

Finally, when a CGI program finishes handling a request, the program terminates. This approach makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data. Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between client requests.

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 high-level utilities. In CGI, you have to do much of this yourself.

Besides, if you already know the Java programming language, why learn CGI too? You're already convinced that Java technology makes for more reliable and reusable code than does Visual Basic, VBScript, or C++. Why go back to those languages for server-side programming?

Powerful

Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI. Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API.
Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations. Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

Portable
Servlets are written in the Java programming language and follow a standard API. Servlets are supported directly by *every* major Web server.
Therefore, servlets written for, say, Macromedia JRun can run virtually unchanged on Apache Tomcat, Microsoft Internet Information Server (with a separate plugin), IBM WebSphere, iPlanet Enterprise Server, Oracle9i AS, or StarNine WebStar.

Inexpensive

A number of free or very inexpensive Web servers are good for development use or deployment of low- or medium-volume Web sites.

Thus, with servlets and JSP you can start with a free or inexpensive server and migrate to more expensive servers with high-performance capabilities or advanced administration utilities only after your project meets initial success.

But CGI require a significant initial investment for the purchase of a proprietary package.

Secure
One of the main sources of weaknesses in traditional CGI is that the programs are often executed by general-purpose operating system shells. So, the CGI programmer must be careful to filter out characters such as back quotes and semicolons that are treated specially by the shell.

A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. For example, in C and C++ it is perfectly legal to allocate a 100-element array and then write into the 999th "element," which is really some random part of program memory. So, programmers who forget to perform this check open up their system to accidental buffer overflow attacks.

Servlets suffer from neither of these problems. Even if a servlet executes a system call (e.g., with Runtime.exec or JNI) to invoke a program on the local operating system, it does not use a shell to do so. And, of course, array bounds checking and other memory protection features are a central part of the Java programming language.

Mainstream
Servlet and JSP technology is supported by servers from Apache, Oracle, IBM, Sybase, BEA, Macromedia, Caucho, Sun/iPlanet, New Atlanta, ATG, Fujitsu, Lutris, Silverstream, the World Wide Web Consortium (W3C), and many others.

Several low-cost plugins add support to Microsoft IIS and Zeus as well. They run on Windows, Unix/Linux, MacOS, VMS, and IBM mainframe operating systems.

Servlets & JSP are the single most popular application of the Java programming language. They are possibly the most popular choice for developing medium to large Web applications.

They are used by the airline industry (most United Airlines and Delta Airlines Web sites), e-commerce (ofoto.com), online banking (First USA Bank, Banco Popular de Puerto Rico), Web search engines/portals (excite.com), large financial sites (American Century Investments), and hundreds of other sites that you visit every day.

THE ROLE OF JSP:

Servlets are Java programs with HTML embedded inside of them. JSP document are HTML pages with Java code embedded inside of them.

Servlets looks mostly like a regular Java class, whereas the JSP looks mostly like a normal HTML page.

Behind the scenes they are the same. In fact, a JSP document is just another way of writing a servlet.

JSP pages get translated into servlets, the servlets get compiled, and it is the servlets that run at request time.

Using these two technology programmer achieves convenience, ease of use, and maintainability.

JSP is focused on simplifying the creation, presentation and maintenance of the HTML. Servlets are best for implementing the business logic and performing complicated operations.

Simple JSP Page to display Current Date & Time of server on client

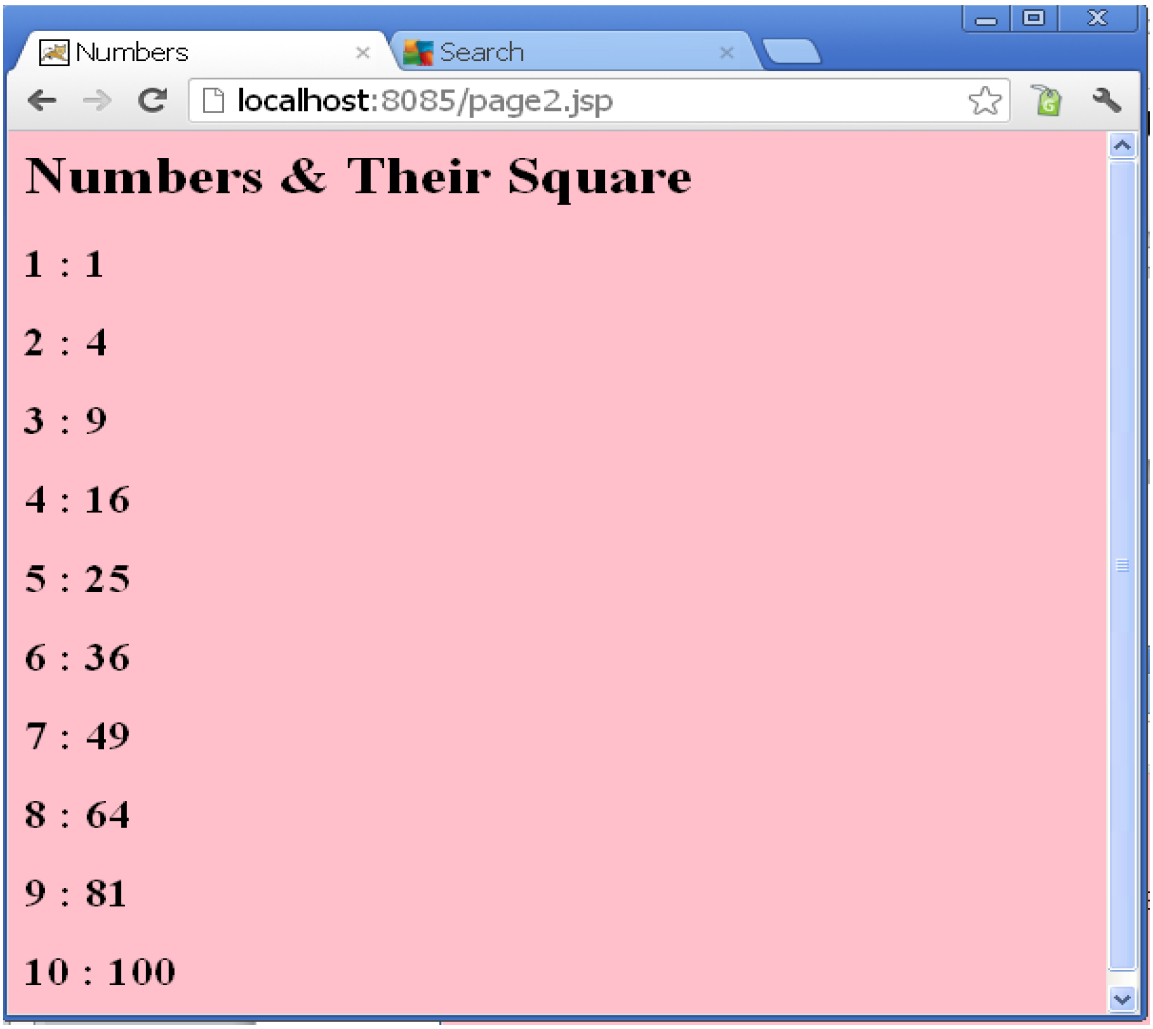
```
<HTML>
<HEAD><TITLE>Welcome to Our Store</TITLE></HEAD>
<BODY BGCOLOR=”pink”>
<H1>Welcome to Our Store</H1>

<P>Current Date & Time of Accessing this Page:
<%
//Java Code to Print Current Date & Time on Page:

out.println(new java.util.Date());

%>
</P>
Regular HTML for rest of online store’s Web page
</BODY>
</HTML>
```

//Output

Simple JSP Page to display 1 to 10 Numbers & Their Square:

```
<HTML>
<HEAD><TITLE>Numbers</TITLE></HEAD>
<BODY BGCOLOR="pink">
<H1>Numbers & Their Square</H1>
<%
//Java Code to Print 1 to 10 numbers & their square
int i;
for (i=1;i<=10;i++)
{
        out.println("<h2> " + i + " : " + (i*i) + "</h2>");
}
%>
</BODY>
</HTML>
```

//Output



INSTALLING & CONFIGURING THE JDK & APACHE TOMCAT:

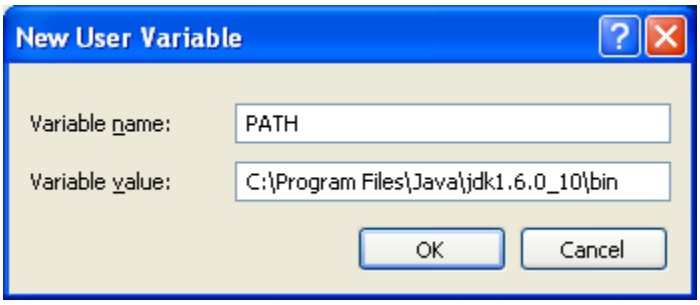
Before you can start learning specific servlet and JSP techniques, you need to have the right software and know how to use it. In this point we have to learn how to obtain, configure, test, and use free versions of all the software needed to run servlets and Java Server Pages (JSP).

The initial setup involves following steps, as outlined below.

1. **Download and install the Java Software Development Kit (SDK).**

This step involves downloading an installation of the Java 2 Platform, Standard Edition and setting your PATH appropriately.

For Windows operating system you can obtain Java 1.6 at http://java.sun.com/. Be sure to download the SDK (Software Development Kit), not just the JRE (Java Runtime Environment)—the JRE is
Used only for executing already compiled Java class files and lacks a compiler.
Once you downloads you will get **jdk-6u10-windows-i586-p.exe File on your computer, double click on it for installation.**
Then installation wizard begins & it will ask the path for installation, keep the default path & install it.
After that you will find java is installed on location **C:\Program Files\Java.**
Then set the PATH variable on environment variable as follows.
Right Click on **My Computer** icon on desktop then➔select **Properties** ➔select **Advanced** tab➔Click on **Environment Variable** Button then➔ Click on **New** button and add the name & values as shows below.

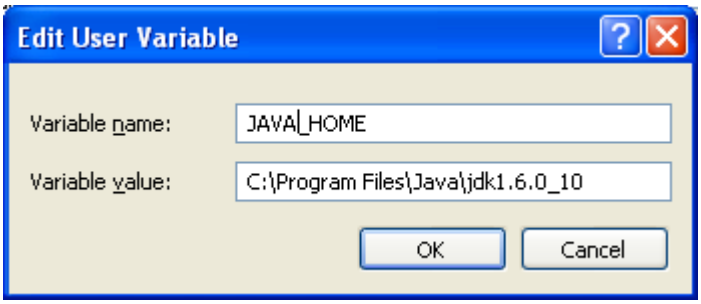


Enter variable Name **PATH** & variable value: **C:\Program Files\Java\jdk1.6.0_10\bin** & click on Ok.

2. **Download a server.** This step involves obtaining a server that implements the servlet & JSP APIs.
   Apache tomcat webserver is most popular web server in world & over 65% of website is hosted on this server.
   Other servers are IIS (Internet Information Server, Oracle HTTP Server, Web Logic Server, XAMPP server are also used for servlet development but we have to apache tomcat web server.
   For Windows operating system you can download the apache web server for development of servlet & JSP pages from the following URL:
   http://jakarta.apache.org/tomcat/
   Once you download a binary version of installation you will get folder named: **apache-tomcat-6.0.10** simply put this folder on C Drive as **C:\apache-tomcat-6.0.10.** This indicates we have installed apache tomcat webserver

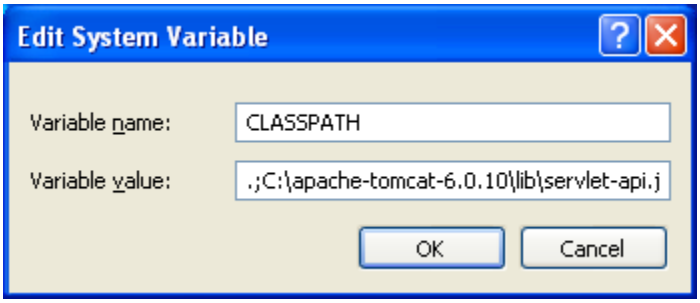
   Configure the server. This step involves telling the server where the SDK is installed by setting the JAVA_HOME variable & changing the port to 80
   You have to create JAVA_HOME as follows:

Right Click on **My Computer** icon on desktop then➔select **Properties** ➔select **Advanced** tab➔Click on **Environment Variable** Button then➔ Click on **New** button and add the name & values as shows below.



3. **Set up your development environment.** This step involves setting your CLASSPATH to include your top-level development directory and the JAR file containing the servlet and JSP classes.

You have to create CLASSPATH as follows:
Right Click on **My Computer** icon on desktop then➔select **Properties** ➔select **Advanced** tab➔Click on **Environment Variable** Button then➔ Click on **New** button and add the name & values as shows below.



CLASSPATH=.;C:\apache-tomcat-6.0.10\lib\servlet-api.jar;C:\apache-tomcat-6.0.10\lib\jsp-api.jar;C:\apache-tomcat-6.0.10\lib\el-api.jar;C:\Servlets+JSP;..;..\..

4. **Test your setup.** This step involves checking the server home page and trying some simple JSP pages and servlets

TESTING YOUR SETUP:

Testing your setup involves checking the server home page and trying some simple JSP pages and servlets.

Testing your setup involves following steps:

1. **Verifying your SDK installation.** Be sure that both java and javac work properly.
2. **Checking your basic server configuration.** Access the server home page, a simple user defined HTML page, and a simple user defined JSP page.
3. **Compiling and deploying some simple servlets.** Try a basic servlet.

First check whether JDK, Apache tomcat web server is properly installed.
It not installed properly follow all the procedure for installation & configuration.
Then check whether the following environment variables are created properly as follows.
Right Click on **My Computer** icon on desktop then➔select **Properties** ➔select **Advanced** tab➔Click on **Environment Variable** Button then➔ creates the follows environment variable.

PATH=C:\Program Files\Java\jdk1.6.0_10\bin
JAVA_HOME=C:\Program Files\Java\jdk1.6.0_10
CLASSPATH=.;C:\apache-tomcat-6.0.10\lib\servlet-api.jar;C:\apache-tomcat-6.0.10\lib\jsp-api.jar;C:\apache-tomcat-6.0.10\lib\el-api.jar;C:\Servlets+JSP;..;..\..

After these settings you have to start the tomcat web server. For that you have to double click on **statup.bat** file available in location: **C:\apache-tomcat-6.0.10\bin**
For testing the setup you have to enter the following URL on web Browser:

http://localhost/ or http://machinename

After that if you get the default page of apache tomcat web server then your installation & configuration is ok

Default location for web application or website in tomcat:
For HTML & JSP Files:
You have to save your HTML & JSP Pages in location:
C:\apache-tomcat-6.0.10\webapps\ROOT

URL for accessing or executing HTML & JSP files:
http://localhost/page1.html or http://machinename/page1.html
http://localhost/page2.jsp or http://machinename/page2.jsp

For servlet class files:
You have to save your servlets class files in location

C:\apache-tomcat-6.0.10\webapps\ROOT\WEB-INF\classes

URL for accessing or executing servlets:
http://localhost/servlet/myservlet or http://machinename/servlet/myservlet

WEB APPLICATIONS: A PREVIEW:

Web Application is a directory or folder contains all the web files (Servlet & JSP Pages) for a particular website or organization.

For creating your web application using servlet & JSP first creates web directory for your files on server & then put all your files into that directory.

The following list summarizes the steps: for creating & previewing your website or application.

1. **Make a directory for your Web application.** HTML and JSP documents go in the directory that you have created, the web.xml file goes in the WEB-INF subdirectory, and servlets and other classes go either in WEB-INF/classes or in a subdirectory of WEB-INF/classes that matches the package name.
2. **Update your CLASSPATH.** Specify the location of your web application directory.
3. **Register the Web application with the server.** Tell the server where the Web application directory is located and what prefix in the URL should be used to invoke the application. For example, with Tomcat, just drop the Web application directory in **C:\apache-tomcat-6.0.10\webapps\ROOT** and then restart the server. The name of the directory becomes the Web application prefix.
4. **Use the designated URL prefix to invoke servlets or HTML/JSP pages from the Web application.** Invoke unpackaged servlets with a default URL of http://*localhost*/***webAppDirName***/servlet/*ServletName*, and packaged servlets with http://*localhost*/***webAppDirName***/servlet/*package-Name.ServletName* , and HTML pages from the top-level Web application directory with http://*host*/***webAppDirName***/*filename*.html.

   For example if you creates a folder or directory named **cocsit** in **C:\apache-tomcat-6.0.10\webapps\ROOT**
   Then your URL for executing or previewing your web application will be
   http://localhost/cocsit/main.jsp ➔for JSP pages
   http://localhost/cocsit/main.html ➔for HTML pages
   http://*localhost*/***cocsit***/servlet/*ServletName* ➔for servlet

BASIC SERVLET STRUCTURE:

Following Program shows a basic servlet structure that handles GET requests.

GET requests, are the type of browser requests for Web pages. A browser generates this request when the user enters a URL on the address line, follows a link from a Web page, or submits an HTML form that either does not specify a METHOD or specifies METHOD="GET".

Servlets can also easily handle POST requests, which are generated when someone submits an HTML form that specifies METHOD="POST".

Table: Basic Servlet Structure

```
import java.io.*; import

javax.servlet.*;

import javax.servlet.http.*;

public class ServletTemplate extends HttpServlet

{

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException

{

// Use "request" to read incoming data explicit & implicit

// (e.g., cookies) and query data from HTML forms.

// Use "response" to specify the HTTP response status

// code and headers (e.g., the content type, cookies).

PrintWriter out = response.getWriter();

// Use "out" to send content to browser.

}
```

Servlets typically extend **HttpServlet** and override **doGet** or **doPost**, depending on whether the data is being sent by **GET** or by **POST**.

If you want a servlet to take the same action for both GET and POST requests, simply have **doGet** call **doPost**, or vice versa.

Both **doGet** and **doPost** take two arguments: an **HttpServletRequest** and an **HttpServletResponse**.

The **HttpServletRequest** lets you get at all of the ***incoming* data**; the class has methods by which you can find out about information such as **form data**, **HTTP request headers**, and the client's hostname.

The **HttpServletResponse** lets you specify *outgoing* information such as response headers (Content-Type, Encoding system, compression technique, Set-Cookie, etc.).

Most importantly, **HttpServletResponse** lets you obtain a **PrintWriter** that you use to send document content back to the client.

For simple servlets, most of the effort is spent in **println** statements that generate the desired page.

Since doGet and doPost throw two exceptions (ServletException and IOException), you are required to include them in the method declaration.

Finally, you must import classes in java.io (for PrintWriter, etc.), javax.servlet (for HttpServlet, etc.), and javax.servlet.http (for HttpServletRequest and HttpServletResponse).

A SERVLET THAT GENERATES PLAIN TEXT:

Following program shows a simple servlet that outputs plain text, with the output shown as below

```
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");
        }
}
```

For testing this program, be sure you have installed & configure JDK & Apache tomcat web server, and you have started web server.

Save the above program on any drive or folder & give the name to that **HelloWorld.java**

If you save in **d:\javaprog** then your path will be **d:\javaprog\HelloWorld.**

Then compile the program on command prompt using **javac** command

D:\java> javac HelloWorld.java

If there is no any error then **HelloWorld.class** file will be created on **D:\java**

Then copy that class file on location: **C:\apache-tomcat-6.0.10\webapps\ROOT\WEB-INF\classes**

Or update the value of your **CLASSPATH** Environment variable as follows:

CLASSPATH=.;C:\apache-tomcat-6.0.10\lib\servlet-api.jar;C:\apache-tomcat-6.0.10\lib\jsp-api.jar;C:\apache-tomcat-6.0.10\lib\el-api.jar;C:\Servlets+JSP;..;..\.. **;D:\javaprog**

After that you can run your above servlet using URL:

http://*localhost*/servlet/*HelloWorld*

Above output shows the servlet being accessed by means of the default URL, with the server running on the local machine.

A SERVLET THAT GENERATES HTML:

Most servlets generate HTML, not plain text. To generate HTML, you have to add three steps as below.

1. Tell the browser that you're sending it HTML.
2. Modify the **println** statements to build a legal Web page.
3. Check your HTML with a formal syntax validator.

For the first step you have to set response as **text/html**. For that you have set content type using function setContentType method, so the following line of code you have add in your servlet.

response.setContentType("text/html");

HTML is the most common kind of document that servlets create, servlet can also create other document types such as Excel spreadsheets (**response.setContentType(“**application/vnd.ms-excel”)), JPEG images (**response.setContentType(“**image/jpeg”)) etc.

You have to call this function before actually returning any of the content with the PrintWriter.

For the second step you have to write HTML code in println statements as shown in following **HelloServlet.java** program,

Program: HelloServlet.java

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)

   throws IOException, ServletException

   {

      response.setContentType("text/html");

      PrintWriter out = response.getWriter();

      out.println("<html>");

      out.println("<head>");
```

```
        out.println("<title>Hello</title>");
        out.println("</head>");
        out.println("<body bgcolor=pink>");
        out.println("<h1>Hello </h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

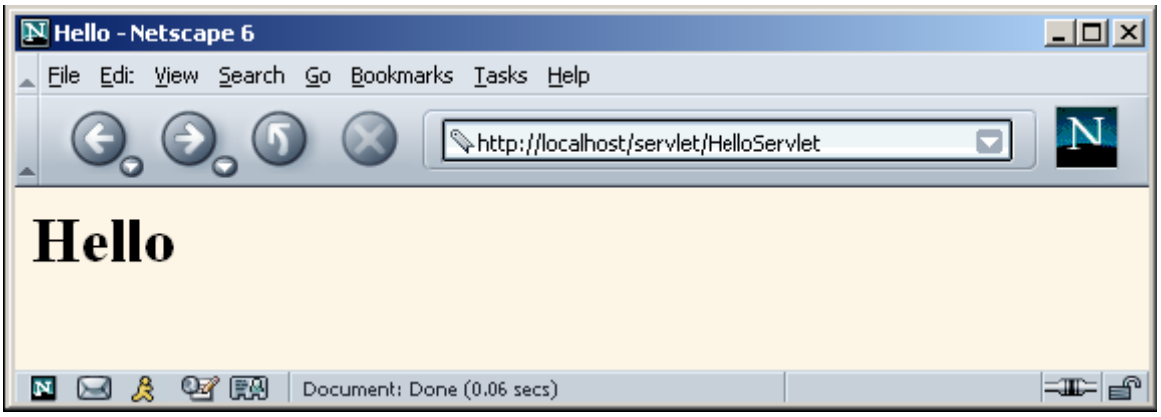


Fig: Result of URL: http://localhost/servlet/HelloServlet

The final step is to check that your HTML has no syntax errors that could cause unpredictable results on different browsers.

SERVLET PACKAGING

In a production environment, multiple programmers can be developing servlets for the same server.

So, placing all the servlets in the same directory results in hard-to-manage collection of classes and risks.

Name conflicts when two developers choose the same name for a servlet or a utility class.

Web application can be large, for that you require the standard Java solution for avoiding name conflicts, and you may want to reuse the code in various servlets, for that you can create your packages & can use it in any servlets.

When you put your servlets in packages & you want to use a class developed in java, then you need to perform the following two additional steps.

1. **Place the files in a subdirectory that matches the planned package name.**
2. **Insert a package statement in the class file.**

Following example shows creating & using package in servlet:

Program **HtmlClass.java** for creating Package

```
//Creating User Defined Package MyPack Contains HtmlClass;


package MyPack;

public class HtmlClass

{

        public String getHtml(String title, String color, String msg)

        {

                String str="<HTML><HEAD><TITLE>" + title +" </TITLE></HEAD>" +

                        "<BODY BGCOLOR=" + color + ">" +

                        "<H1>" +msg +"</H1>";

                return str;

        }

}
```

Use the command for compiling program;

D:\JavaProg>javac –d d:\JavaProg HtmlClass.java

Program: Servlet **MyServlet.java** for using Package MyPack

Use the command for compiling program;

```
//Creating Servlet MyServlet that uses package-MyPack & Class-HtmlClass in MyPack package
MyPack;
import java.io.*; import
javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse response) throws
   IOException, ServletException
   {
      response.setContentType("text/html"); PrintWriter out =
      response.getWriter();
      HtmlClass h=new HtmlClass(); // Class Defined in Package MyPack
      String str=h.getHtml("Cocsit","pink","College of Computer Science & IT Latur" );
      //Above function is in HtmlClass from MyPack
      out.println(str);
      out.println("</body>");
      out.println("</html>");
   }
```

D:\JavaProg>javac –d d:\JavaProg MyServlet.java

Fig: Use URL : **http://localhost:8085/servlet/MyPack.MyServlet**

Output.

THE SERVLET LIFE CYCLE

The Servlet life cycle is the time span between servlet created & destroyed. Servlet life demonstrates how servlets are created and destroyed, and how and when the various methods are invoked.

When the servlet is first created, its **init** method is invoked, so init is where you put one-time setup code.

After this, each user request results in a thread that calls the **service** method of the previously created instance. Multiple concurrent requests normally result in multiple threads calling service simultaneously.

The service method then calls **doGet, doPost**, or another **do*Xxx*** method, depending on the type of HTTP request it received.

Finally, if the server decides to unload a servlet, it first calls the servlet's **destroy** method.

The service Method

Each time the server receives a request for a servlet; the server creates a new thread and calls service.

The service method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc., as appropriate.

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified.

A POST request results from an HTML form that specifically lists POST as the METHOD.

Other HTTP requests are generated only by custom clients.

Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override service directly rather than implementing both doGet and doPost. This is not a good idea. Instead, just have doPost call doGet (or vice versa), as below.

```
public void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
// Servlet code
}
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
doGet(request, response);
}
```

The doGet, doPost, and doXxx Methods

These methods contain the real meat of your servlet. Ninety-nine percent of the time, you only care about GET or POST requests, so you override doGet and/or doPost. However, if you want to, you can also override doDelete for DELETE requests, doPut for PUT, doOptions for OPTIONS, and doTrace for TRACE.

The init Method

Most of the time, your servlets deal only with per-request data, and doGet or doPost are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The init method is designed for this case;

It is called when the servlet is first created, and *not* called again for each user request. So, it is used for one-time initializations, just as with the init method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

The init method definition looks like this:

```
public void init() throws ServletException

{

// Initialization code...

}
```

The destroy Method

The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator or perhaps because the servlet is idle for a long time. Before it does, however, it calls the servlet's destroy method. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

Following Program demonstrates the use of init function for displaying 10 random Numbers as Lucky Lottery Numbers:

Numbers will be initialized only once.

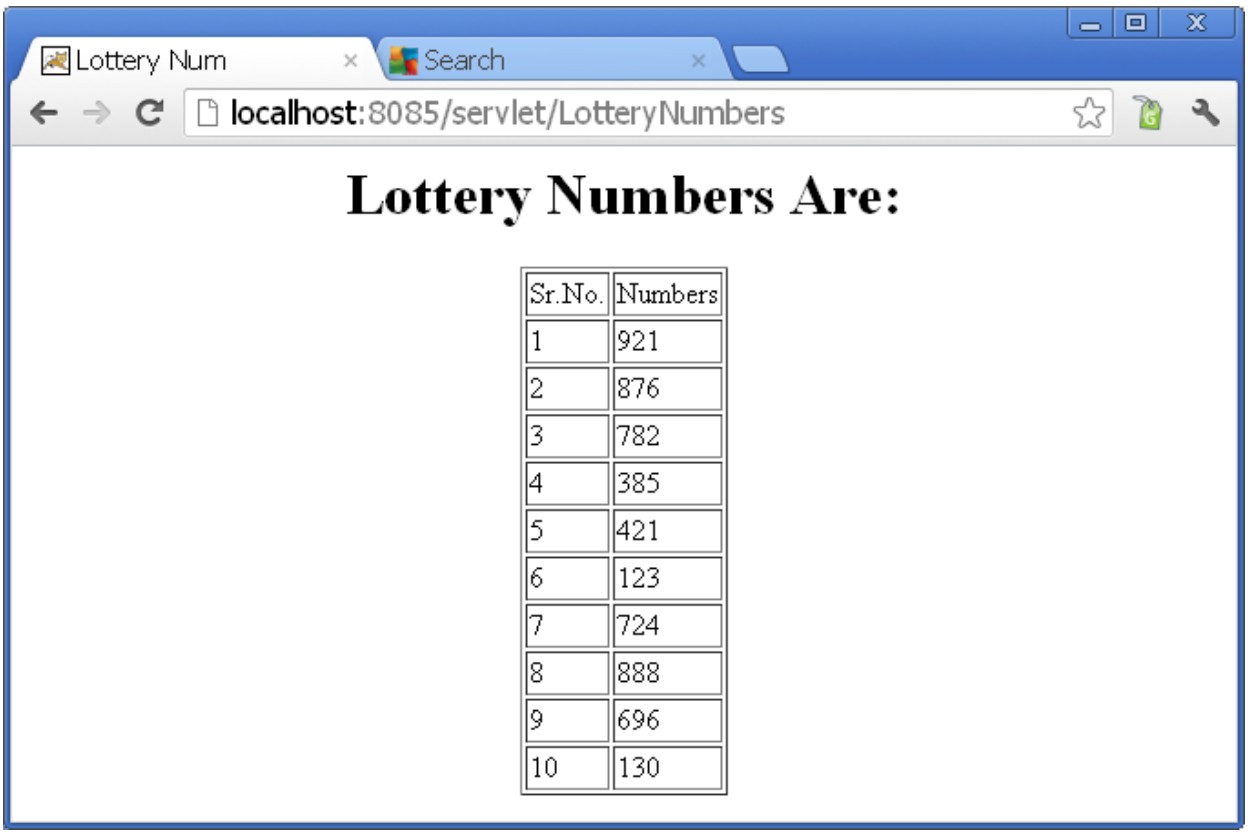
Program: **LotteryNumbers.java**

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class LotteryNumbers extends HttpServlet

{

int num[]=new int [10];

public void init() throws ServletException

{
```

```
        //Loop will store 10 Random Numbers in Array num
        // Used for one time initialization
        for(int i=0;i<10;i++)
        {
                num[i]=(int)(Math.random()*1000);
                //function for getting random number
        }
}
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Lottery Num</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1 align=center> Lottery Numbers Are:</h1>");
     out.println("<table border=1 align=center>");
     out.println("<tr><td>Sr.No.</td><td> Numbers </td></tr>");
     for(int i=0;i<10;i++)
     {
        out.println("<tr><td>"+(i+1) + "</td><td>" + num[i] +"</td></tr>");
      }
```

```
        out.println("</table>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Fig: Output of URL: **http://localhost:8085/servlet/LotteryNumbers**



THE SINGLE THREAD MODEL INTERFACE:

Normally, the system makes a single instance of your servlet and then creates a new thread for each user request. This means that if a new request comes in while a previous request is still executing, multiple threads can concurrently be accessing the same servlet object.

Therefore, your doGet and doPost methods must be careful to synchronize access to fields and other shared data (if any) since multiple threads may access the data simultaneously.

Note that local variables are not shared by multiple threads, and thus need no special protection.

You can prevent multithreaded access by having your servlet implement the SingleThreadModel interface, as below.

```
public class YourServlet extends HttpServlet
implements SingleThreadModel
{
}
```

If you implement this interface, the system guarantees that there is never more than one request thread accessing a single instance of your servlet. In most cases, it does so by queuing all the requests and passing them one at a time to a single servlet instance.

Although SingleThreadModel prevents concurrent access in principle, in practice there are two reasons why it is usually a poor choice.

First, synchronous access to your servlets can significantly hurt performance if your servlet is accessed frequently. When a servlet waits for I/O, the server cannot handle pending requests for the same servlet. So, think twice before using the SingleThreadModel approach. Instead, consider synchronizing only the part of the code that manipulates the shared data.

The second problem with SingleThreadModel stops from the fact that the specification permits servers to use pools of instances instead of queueing up the requests to a single instance.

For example, consider the following servlet that attempts to assign unique user IDs to each client (unique until the server restarts, that is).

It uses an instance variable (field) called nextID to keep track of which ID should be assigned next, and uses the following code to output the ID.

```
String id = "User-ID-" + nextID;

out.println("<H2>" + id + "</H2>");

nextID = nextID + 1;
```

You started the server. You repeatedly accessed the servlet with http://localhost/servlet/.UserIDs.

Every time you accessed it, you got a different value.

The problem occurs only when there are multiple simultaneous accesses to the servlet. Even then, it occurs only once in a while. But, in a few cases, the first client could read the nextID field and have its thread preempted before it incremented the field. Then, a second client could read the field and get the same value as the first client.

The proper solution for this is any one of the following three as per your convinence

1. **Shorten the race.**
   Remove the third line of the code snippet and change the first line to the following.

```
String id = "User-ID-" + nextID++;
```

2. **Use SingleThreadModel.**
   Change the servlet class definition to the following.

```
public class UserIDs extends HttpServlet implements SingleThreadModel

{

}
```

The server implements SingleThreadModel then all the requests will queueing up, but performance will low if there is a lot of concurrent access.

3. **Synchronize the code explicitly.**
   Use the standard synchronization construct of the Java programming language. Start a synchronized block just before the first access to the shared data, and end the block just after the last update to the data, as follows.

```
synchronized(this)

{

String id = "User-ID-" + nextID;
```

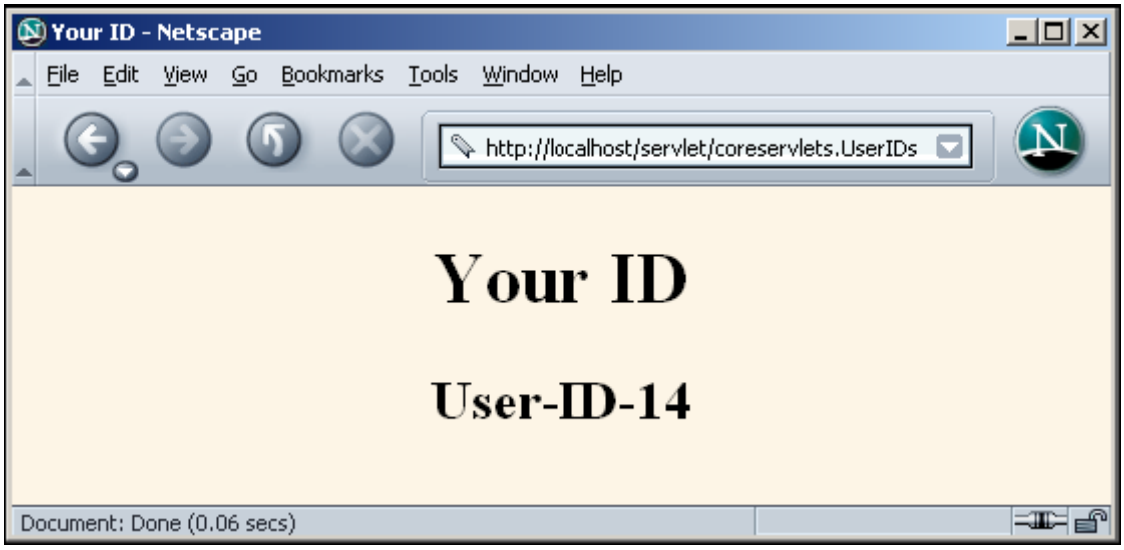
```
out.println("<H2>" + id + "</H2>");

nextID = nextID + 1;

}
```

This technique tells the system that, once a thread has entered the above block of code (or any other synchronized section labeled with the same object reference), no other thread is allowed in until the first thread exits.

This is the solution you have always used in the Java programming language.

```
package coreservlets;

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

/** Servlet that attempts to give each user a unique user ID. However, because it fails to synchronize access to the nextID field, it suffers from race conditions: two users could get the same ID.

*/

public class UserIDs extends HttpServlet

{

    private int nextID = 0;

    public void doGet(HttpServletRequest request, HttpServletResponse response)

    throws ServletException, IOException

    {

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    String title = "Your ID";

    out.println("<HTML> " +

    "<HEAD><TITLE>User IDs </TITLE></HEAD>" +
```

```
        "<CENTER>" +
        "<BODY BGCOLOR=\"pink\">" +
        "<H1> User IDs </H1>");
        String id = "User-ID-" + nextID;
        out.println("<H2>" + id + "</H2>");
        nextID = nextID + 1;
        out.println("</BODY></HTML>");
        }
}
```



SERVLET DEBUGGING:

Debugging is the procedure of testing the program to find all possible errors.

Debugging servlets can be tricky because you don't execute them directly. Instead, you trigger their execution by means of an HTTP request, and they are executed by the Web server. This remote execution makes it difficult to insert break points or to read debugging messages and stack traces. So, approaches to servlet debugging differ somewhat from those used in general development.

Following are 10 general method used for servlet debugging:

1. Use print statements.

System.out.println statement is used to print the intermediate result of program on console window of tomcat web server. This you can use to find out the reason of errors in program.

2. Use an integrated debugger in your IDE.

Many integrated development environments (IDEs) have sophisticated debugging tools that can be integrated with your servlet and JSP. The Enterprise editions of IDEs like Borland JBuilder, Oracle JDeveloper, IBM WebSphere Studio, Eclipse, BEA WebLogic Studio, Sun ONE Studio, etc., typically let you insert breakpoints, trace method calls, and so on. Some will even let you connect to a server running on a remote system.

3. Use the log file.

The HttpServlet class has a method called log that lets you write information into a logging file on the server. Reading debugging messages from the log file is a bit less convenient than watching them directly from a window as with the two previous approaches.

4. Use Apache Log4J.

Log4J is a package from the Apache Jakarta Project—the same project that manages Tomcat and Struts. With Log4J, you semi-permanently insert debugging statements in your code and use an XML-based configuration file to control which are invoked at request time. Log4J is fast, flexible, convenient, and becoming more popular by the day.

5. Write separate classes.

One of the basic principles of good software design is to put commonly used code into a separate function or class so you don't need to keep rewriting it. That principle is even more important when you are writing servlets, since these separate classes can often be tested independently of the server. You can even write a test routine, with a main, that can be used to generate hundreds or thousands of test cases for your routines—not something you are likely to do if you have to submit each test case by hand in a browser.

6. Plan ahead for missing or malformed data.

Every time you process data that comes directly or indirectly from a client, be sure to consider the possibility that it was entered incorrectly or omitted altogether.

7. Look at the HTML source.

If the result you see in the browser looks odd, choose View Source from the browser's menu. Sometimes a small HTML error like <TABLE> instead of </TABLE> can prevent much of the page from being viewed.

8. Look at the request data separately.

Servlets read data from the HTTP request, construct a response, and send it back to the client. If something in the process goes wrong, you want to discover if the cause is that the client is sending the wrong data or that the servlet is processing it incorrectly. So check the request data separately.

9. Look at the response data separately.

Once you look at the request data separately, you'll want to do the same for the response data, so check the response data separately.

10. Stop and restart the server.

Servers are supposed to keep servlets in memory between requests, not reload them each time they are executed. However, most servers support a development mode in which servlets are supposed to be automatically reloaded whenever their associated class files changes. If your problems not solved then try restarting the server.