UNIT – IV – JSP NOTES

JSP Overview

JSP is the latest Java technology for web application development, and is based on the servlet technology introduced in the previous chapter. While servlets are great in many ways, they are generally reserved for programmers. In this chapter, we look at the problems that JSP technology solves, the anatomy of a JSP page, the relationship between servlets and JSP, and how a JSP page is processed by the server. In any web application, a program on the server processes requests and generates responses. In a simple one-page application, such as an online bulletin board, you don't need to be overly concerned about the design of this piece of code; all logic can be lumped together in a single program. But when the application grows into something bigger (spanning multiple pages, with more options and support for more types of clients) it's a different story. The way your site is designed is critical to how well it can be adapted to new requirements and continues to evolve. The good news is that JSP technology can be used in all kinds of web applications, from the simplest to the most complex. Therefore, this chapter also introduces the primary concepts in the design model recommended for web applications, and the different roles played by JSP and other Java technologies in this model.

The Problem with Servlets

In many Java servlet-based applications, processing the request and generating the response are both handled by a single servlet class. A example servlet looks like this:

```
public class OrderServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter( );
if (isOrderInfoValid(request)) {
saveOrderInfo(request);
out.println("<html>");
out.println(" <head>");
out.println(" <title>Order Confirmation</title>");
out.println(" </head>");
out.println(" <body>");
out.println(" <h1>Order Confirmation</h1>");
renderOrderInfo(request);
out.println(" </body>");
out.println("</html>");
}
```

If you're not a programmer, don't worry about all the details in this code. The point is that the servlet contains request processing and business logic (implemented by methods such as isOrderInfoValid( ) and saveOrderInfo( )) and also generates the

response HTML code, embedded directly in the servlet code using println( ) calls. A more structured servlet application isolates different pieces of the processing in various reusable utility classes, and may also use a separate class library for generating the actual HTML elements in the response. But even so, the pure servlet-based approach still has a few problems:

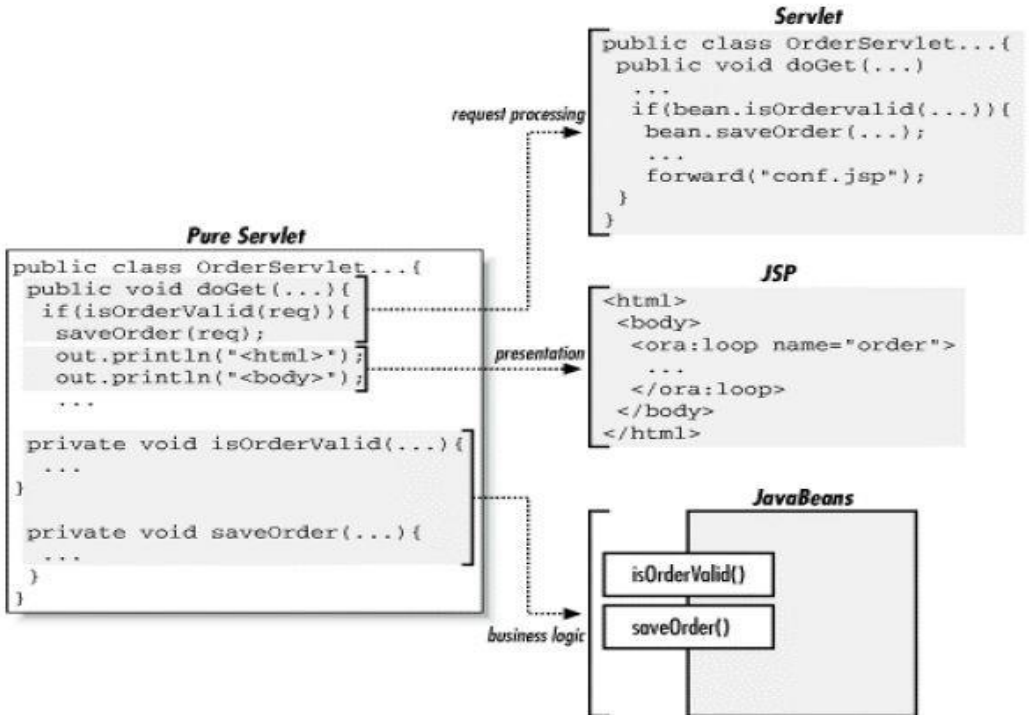
· Detailed Java programming knowledge is needed to develop and maintain all aspects of the application, since the processing code and the HTML elements are lumped together.

· Changing the look and feel of the application, or adding support for a new type of client (such as a WML client), requires the servlet code to be updated and recompiled.

· It's hard to take advantage of web page development tools when designing the application interface. If such tools are used to develop the web page layout, the generated HTML must then be manually embedded into the servlet code, a process that is time-consuming, error-prone, and extremely boring.

Adding JSP to the puzzle lets you solve these problems by separating the request processing and business logic code from the presentation, as illustrated in Figure 3.1. Instead of embedding HTML in the code, you place all static HTML in JSP pages, just as in a regular web page, and add a few JSP elements to generate the dynamic parts of the page. The request processing can remain the domain of servlet programmers, and the business logic can be handled by JavaBeans and Enterprise JavaBeans (EJB) components.

Figure 3.1. Separation of request processing, business logic, and presentation



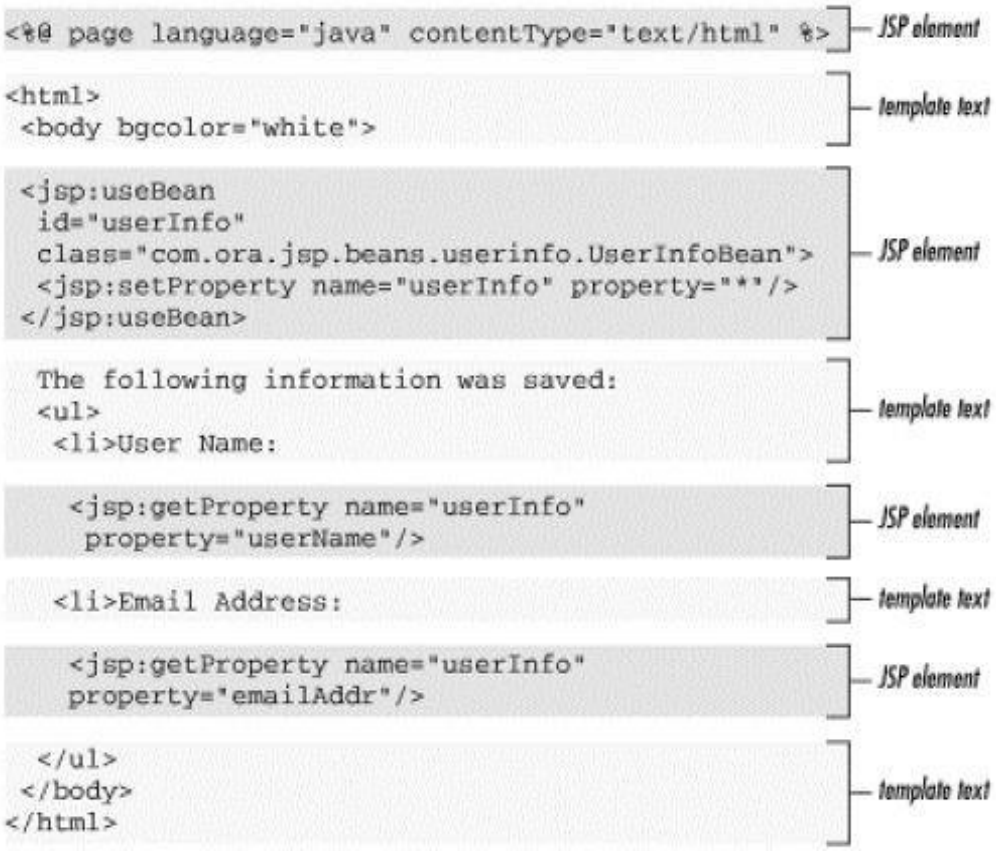
As I mentioned before, separating the request processing and business logic from presentation makes it possible to divide the development tasks among people with different skills. Java programmers implement the request processing and business logic pieces, web page authors implement the user interface, and both groups can use best-of-breed development tools for the task at hand. The result is a much more productive

development process. It also makes it possible to change different aspects of the application independently, such as changing the business rules without touching the user interface. This model has clear benefits even for a web page author without programming skills who is working alone. A page author can develop web applications with many dynamic features, using generic Java components provided by open source projects or commercial companies.

The Anatomy of a JSP Page

A JSP page is simply a regular web page with JSP elements for generating the parts of the page that differ for each request, as shown in Figure 3.2. Everything in the page that is not a JSP element is called *template text*. Template text can really be any text: HTML, WML, XML, or even plain text. Since HTML is by far the most common web page language in use today, most of the descriptions and examples in this book are HTML-based, but keep in mind that JSP has no dependency on HTML; it can be used with any markup language. Template text is always passed straight through to the browser.

Figure 3.2. Template text and JSP elements



When a JSP page request is processed, the template text and the dynamic content generated by the JSP elements are merged, and the result is sent as the response to the browser.

JSP Elements

There are three types of elements with JavaServer Pages: *directive*, *action*, and *scripting* elements. The directive elements, shown in Table 3.1, are used to specify information about the page itself that remains the same between page requests, for example, the scripting language used in the page, whether session tracking is required, and the name of a page that should be used to report errors, if any.

Table 3.1, Directive Elements

Element	Description
<%@ page ... %>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements
<%@ include ... %>	Includes a file during the translation phase
<%@ taglib ... %>	Declares a tag library, containing custom actions, used in the page

Element Description
<%@ page ... %> Defines page-dependent attributes, such as scripting language, error page, and buffering requirements
<%@ include ... %> Includes a file during the translation phase
<%@ taglib ... %> Declares a tag library, containing custom actions, used in the page Action elements typically perform some action based on information that is required at the exact time the JSP page is requested by a client. An action element can, for instance, access parameters sent with the request to do a database lookup. It can also dynamically generate HTML, such as a table filled with information retrieved from an external system.

The JSP specification defines a few standard action elements, listed in Table 3.2, and includes a framework for developing custom action elements. A custom action element can be developed by a programmer to extend the JSP language. The examples in this book use custom actions for database access, internationalization, access control, and more.

Table 3.2, Standard Action Elements

Element	Description
<jsp:useBean>	Makes a JavaBeans component available in a page
<jsp:getProperty>	Gets a property value from a JavaBeans component and adds it to the response
<jsp:setProperty>	Sets a JavaBeans property value
<jsp:include>	Includes the response from a servlet or JSP page during the request processing phase
<jsp:forward>	Forwards the processing of a request to a servlet or JSP page
<jsp:param>	Adds a parameter value to a request handed off to another servlet or JSP page using <jsp:include> or <jsp:forward>
<jsp:plugin>	Generates HTML that contains the appropriate client browser-dependent elements (OBJECT or EMBED) needed to execute an Applet with the Java Plugin software

Element Description

<jsp:useBean> Makes a JavaBeans component available in a page

<jsp:getProperty> Gets a property value from a JavaBeans component and adds it to the response

<jsp:setProperty> Sets a JavaBeans property value

<jsp:include> Includes the response from a servlet or JSP page during the request processing phase

<jsp:forward> Forwards the processing of a request to a servlet or JSP page

<jsp:param> Adds a parameter value to a request handed off to another servlet or JSP page using <jsp:include> or <jsp:forward>

<jsp:plugin>

Generates HTML that contains the appropriate client browser-dependent elements (OBJECT or EMBED) needed to execute an Applet with the Java Plugin software Scripting elements, shown in Table 3.3, allow you to add small pieces of code to a JSP page, such as an if statement to generate different HTML depending on a certain condition. Like actions, they are also executed when the page is requested. You should use scripting elements with extreme care: if you embed too much code in your JSP pages, you will end up with the same kind of maintenance problems as with servlets embedding HTML.

Table 3.3, Scripting Elements

Element	Description
<% ... %>	Scriptlet, used to embed scripting code.
<%= ... %>	Expression, used to embed Java expressions when the result shall be added to the response. Also used as runtime action attribute values.
<%! ... %>	Declaration, used to declare instance variables and methods in the JSP page implementation class.

Element Description

<% ... %> Scriptlet, used to embed scripting code.

<%= ... %> Expression, used to embed Java expressions when the result shall be added to the response. Also used as runtime action attribute values.

<%! ... %> Declaration, used to declare instance variables and methods in the JSP page implementation class.

JSP elements, such as action and scripting elements, are often used to work with *JavaBeans* . Put succinctly, a JavaBeans component is a Java class that complies with certain coding conventions. JavaBeans are typically used as containers for information that describes application entities, such as a customer or an order. We'll cover each of these element types, as well as JavaBeans, in the following chapters.

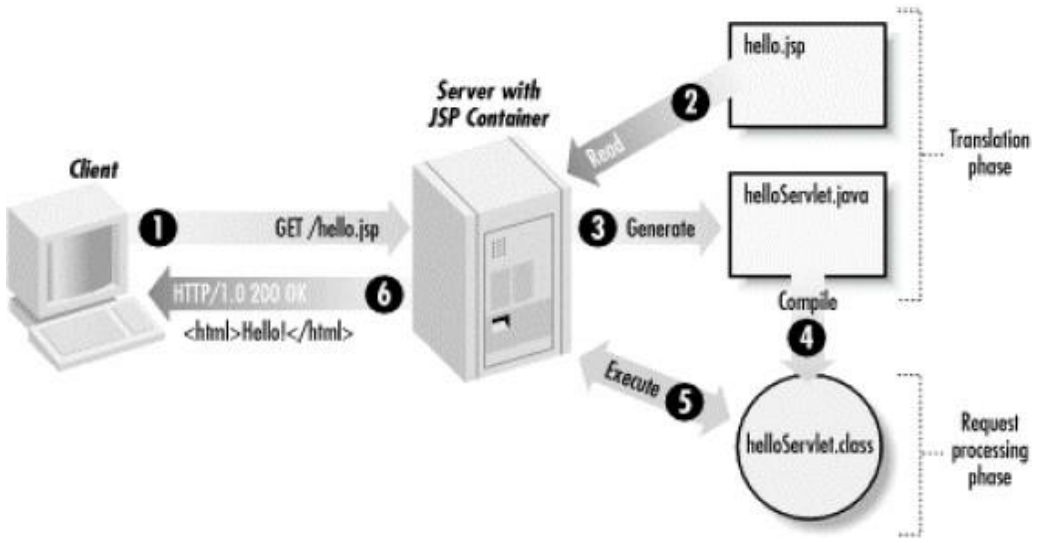
JSP Processing

A JSP page cannot be sent as-is to the browser; all JSP elements must first be processed by the server. This is done by turning the JSP page into a servlet, and then executing the servlet. Just as a web server needs a servlet container to provide an interface to servlets, the server needs a JSP container to process JSP pages. The JSP container is often implemented as a servlet configured to handle all requests for JSP pages. In fact, these two containers - a servlet container and a JSP container - are often combined into one package under the name *web container* (as it is referred to in the J2EE documentation).

A JSP container is responsible for converting the JSP page into a servlet (known as the *JSP page implementation class*) and compiling the servlet. These two steps form the *translation phase* . The JSP container automatically initiates the translation phase for a page when the first request for the page is received. The translation phase takes a bit of time, of course, so a user notices a slight delay the first time a JSP page is requested. The translation phase can also be initiated explicitly; this is referred to as *precompilation* of a JSP page. Precompiling a JSP page avoids hitting the user with this delay.

The JSP container is also responsible for invoking the JSP page implementation class to process each request and generate the response. This is called the *request processing phase*. The two phases are illustrated in Figure 3.3.

Figure 3.3. JSP page translation and processing phases



As long as the JSP page remains unchanged, any subsequent processing goes straight to the request processing phase (i.e., it simply executes the class file). When the JSP page is modified, it goes through the translation phase again before entering the request processing phase. So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. And, except for the translation phase, a JSP page is handled exactly like a regular servlet: it's loaded once and called repeatedly, until the server is shut down.

Let's look at a simple example of a servlet. In the tradition of programming books for as far back as anyone cares to remember, we start with an application that just writes Hello World, but this time we will add a twist: our application will also show the current time on the server. Example 3.1 shows a hand-coded servlet with this functionality.

Example 3.1. Hello World Servlet

```
public class HelloWorldServlet implements Servlet {
public void service(ServletRequest request,
ServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter( );
out.println("<html>");
out.println(" <head>");
out.println(" <title>Hello World</title>");
out.println(" </head>");
out.println(" <body>");
out.println(" <h1>Hello World</h1>");
out.println(" It's " + (new java.util.Date( ).toString( )) +
" and all is well.");
out.println(" </body>");
out.println("</html>");
}
}
```

As before, don't worry about the details if you're not a Java programmer. What's important here is that the service( ) method is the method called by the servlet container every time the servlet is requested, as described in Chapter 2. The method generates all HTML code, using the println( ) method to send the strings to the browser. Note that there's no way you could use a web development tool to develop this type of embedded HTML, adjust the layout with immediate feedback, verify that links are intact, etc. This example is so simple that it doesn't really matter, but imagine a complex page with tables, aligned images, forms, some JavaScript code, etc., and you see the problem.

Also note the following lines, which add the current date and time to the response (shown in Figure 3.4):

```
out.println(" It's " + (new java.util.Date( ).toString( ))+ " and all is well.");
```

Figure 3.4. The output from the Hello World servlet

Example 3.2 shows a JSP page that produces the same result as the Hello World servlet.

Example 3.2. Hello World JSP Page

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1>Hello World</h1>
It's <%= new java.util.Date( ).toString( ) %> and all is well.
</body>
</html>
```

This is as simple as it gets. A JSP page is a regular HTML page, except that it may also contain JSP elements like the highlighted element in this example. This element inserts the same Java code in the page as was used in the servlet to add the current date and time. If you compare this JSP page to the corresponding servlet, you see that the JSP page can be developed using any web page editor that allows you to insert extra, non-HTML elements. And the same tool can later be used to easily modify the layout. This is a great advantage over a servlet with embedded HTML. The JSP page is automatically turned into a servlet the first time it's requested, as described earlier. The generated servlet looks something like in Example 3.3.

Example 3.3. Servlet Generated from JSP Page

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import org.apache.jasper.*;
import org.apache.jasper.runtime.*;
public class _0005chello_0002ejsphello_jsp_1 extends HttpJspBase {
public void _ jspService(HttpServletRequest request,
HttpServletResponse response)
throws IOException, ServletException {
JspFactory _ jspxFactory = null;
```

```
PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
String _value = null;
try {
_ jspxFactory = JspFactory.getDefaultFactory( );
response.setContentType("text/html");
pageContext = _ jspxFactory.getPageContext(this, request,
response,"", true, 8192, true);
application = pageContext.getServletContext( );
config = pageContext.getServletConfig( );
session = pageContext.getSession( );
out = pageContext.getOut( );
out.write("<HTML>\r\n <HEAD>\r\n <TITLE>" +
"Hello World</TITLE>\r\n </HEAD>\r\n" +
" <BODY>\r\n <H1>Hello World</H1>\r\n" +
" It's ");
out.print( new java.util.Date( ).toString( ) );
out.write(" and all is well.\r\n </BODY>\r\n" +
"</HTML>\r\n");
} catch (Exception ex) {
if (out.getBufferSize( ) != 0)
out.clear( );
pageContext.handlePageException(ex);
} finally {
out.flush( );
_ jspxFactory.releasePageContext(pageContext);
}
}
}
```

The generated servlet in Example 3.3 looks a lot more complex than the hand-coded version in Example 3.1. That's because a number of objects you can use in a JSP page must always be initialized (the hand-coded version doesn't need this generic initialization). These details are not important now; programming examples later in the book will show you how to use all objects of interest. Instead, you should note that the servlet generated from the JSP page is a regular servlet. The _jspService() method corresponds to the service( ) method in the hand-coded servlet; it's called every time the page is requested. The request and response objects are passed as arguments to the method, so the JSP page has access to all the same information as does a regular servlet. This means it can read user input passed as request parameters, adjust the response based on header values (like the ones described in Chapter 2), get access to the session state,

etc. - just like a regular servlet. The highlighted code section in Example 3.3 shows how

the static HTML from the JSP page in Example 3.2 has been embedded in the resulting code. Also note that the Java code to retrieve the current date and time has been inserted in the servlet as-is. By letting the JSP container convert the JSP page into a servlet that combines code for adding HTML to the response with small pieces of Java code for dynamic content, you get the best of both worlds. You can use familiar web page development tools to design the static parts of the web page, drop in JSP elements that generate the dynamic parts, and still enjoy all the benefits of servlets.

Client-Side Versus Server-Side Code

Page authors who have some experience developing client-side scripts using JavaScript (ECMAScript) or VBScript can sometimes get a bit confused when they start to use a server-side technology like JSP. Client-side scripts, embedded in <script> elements, execute in the browser. These types of scripts are often linked to a form element such as a selection list. When the user selects an item in the list, the associated script is executed, perhaps populating another selection list with appropriate choices. Since all this code is executed by the browser, the client-side script provides immediate feedback to the user.

Server-side code, like action and scripting elements in a JSP page, executes on the server. Recall from Chapter 2 that the browser must make a request to the server to execute a JSP page. The corresponding JSP code is then used to produce a dynamic response. This brings up an important point: there's no way a client-side script can directly call an individual Java code segment in the JSP page. A client-side script can ask the browser to make a request for the complete page, but it can't process the response and use it to do something such as populate a selection list with the data. It is possible, although not very efficient, to link a user action to a client-side script, invoking an applet that in turn makes a request to a servlet or JSP page. The applet can then read the response and cause some dynamic action in the web browser. This approach may be reasonable on a fast intranet, but you probably won't be happy with the response times if you tried it on the Internet during peak hours. The reason is that the HTTP request/response model was never intended to be used for this type of incremental user interface update. Consequently, there's a great deal of overhead involved. If you still want to do this, be careful not to open up a security hole. For instance, if you develop an applet that can send any SQL statement to a servlet and get the query result back, you have made it possible for anyone to access all data in your database (that is accessible to the servlet), not just the data that your applet asks for.

Client-side and server-side code can, however, be combined with good results. You can embed client side scripts as template text in your JSP pages, or generate it dynamically with actions or scripting elements. But keep in mind that it's still client-side code; the fact that it's generated by a JSP page doesn't change anything. A common use of client-side code is to validate user form input. Doing the validation with client-side code gives the user faster feedback about invalid input and reduces the load on the server. But don't forget that client-side scripting is not supported in all browsers, and even if it is, the user may have disabled the execution of scripts. Therefore, you should always perform

input validation on the server as well. Instead of using client-side scripts, you can of

course use a Java applet to provide a more interactive user interface. Ideally the applet is self-contained; in other words, it doesn't have to talk to the server at all in order to present a user-friendly interface. If it needs to communicate with the server, however, it can do so using a far more efficient protocol than HTTP. *Java Servlet Programming* by Jason Hunter and William Crawford (O'Reilly) includes a chapter about different applet communication options.

JSP Application Design with MVC

JSP technology can play a part in everything from the simplest web application, such as an online phone list or an employee vacation planner, to full-fledged enterprise applications, such as a human resource application or a sophisticated online shopping site. How large a part JSP plays differs in each case, of course. In this section, we introduce a design model suitable for both simple and complex applications called Model-View-Controller (MVC).

MVC was first described by Xerox in a number of papers published in the late 1980s. The key point of using MVC is to separate components into three distinct units: the Model, the View, and the Controller. In a server application, we commonly classify the parts of the application as: business logic, presentation, and request processing. *Business logic* is the term used for the manipulation of an application's data, i.e., customer, product, and order information. *Presentation* refers to how the application is displayed to the user, i.e., the position, font, and size. And finally, *request processing* is what ties the business logic and presentation parts together. In MVC terms, the Model corresponds to business logic and data, the View to the presentation logic, and the Controller to the request processing.

Why use this design with JSP? The answer lies primarily in the first two elements. Remember that an application data structure and logic (the Model) is typically the most stable part of an application, while the presentation of that data (the View) changes fairly often. Just look at all the face-lifts that web sites have gone through to keep up with the latest fashion in web design. Yet, the data they present remains the same.

Another common example of why presentation should be separated from the business logic is that you may want to present the data in different languages or present different subsets of the data to internal and external users. Access to the data through new types of devices, such as cell phones and Personal Digital Assistants (PDAs), is the latest trend. Each client type requires its own presentation format. It should come as no surprise, then, that separating business logic from presentation makes it easier to evolve an application as the requirements change; new presentation interfaces can be developed without touching the business logic.

This MVC model is used for most of the examples in this book. In Part II, JSP pages are used as both the Controller and the View, and JavaBeans components are used as the Model.

Setting Up the JSP Environment

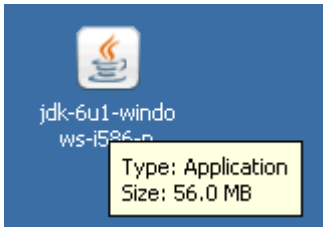
All examples were developed and tested with the JSP reference implementation, known as the Apache Tomcat server, which is developed by the Apache Jakarta project. In this chapter you will learn how to install the Tomcat server and add a web application containing all the examples used in this book. You can, of course, use any web server that supports JSP 1.1, but Tomcat is a good server for development and test purposes. You can learn more about the Jakarta project and Tomcat, as well as how you can participate in the development, at the Jakarta web
site: http://jakarta.apache.org.

Installing the Java Software Development Kit (JSDK or JDK)

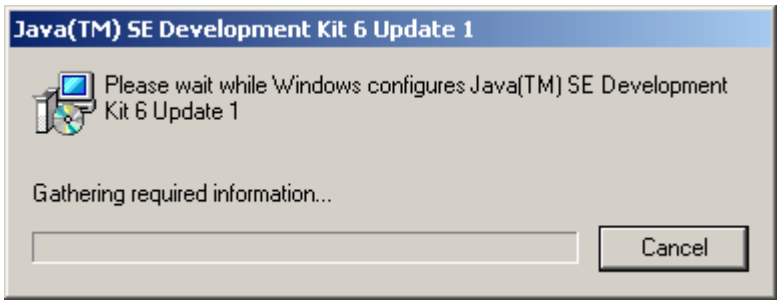
The latest version of jdk is 6 (update 1) at the time of writing the tutorial. You can download the latest version of jdk from http://java.sun.com/javase/downloads/index.jsp. Once you download the exe file you can now install it. Just follow as mentioned:

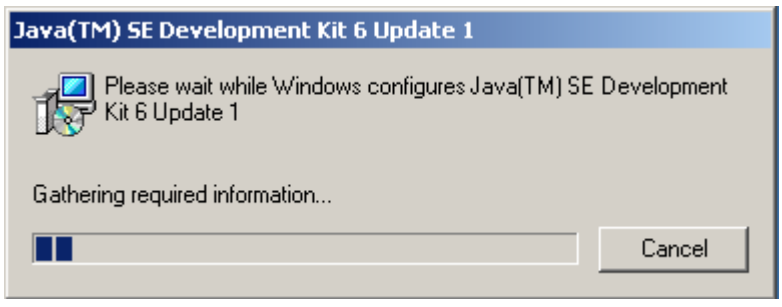
To install the jdk, double click on the downloaded exe file (jdk-6u1-windows-i586-p.exe)

Step 1. Double Click the icon of downloaded exe.

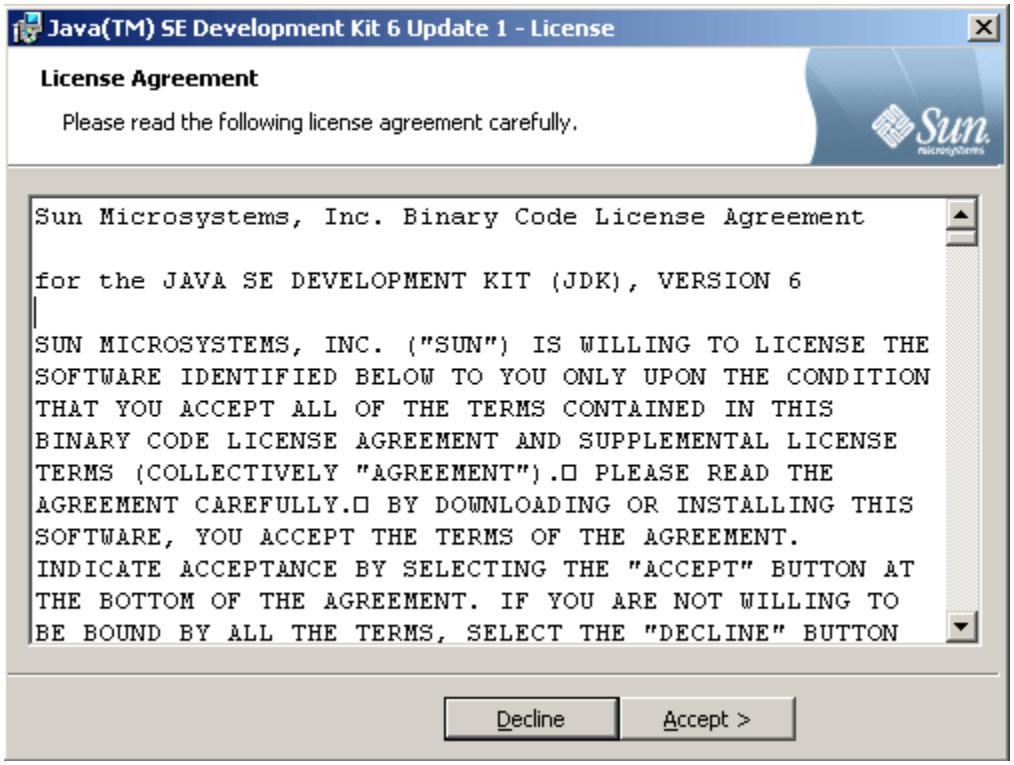


You will see jdk 6 update 1 window as shown below.

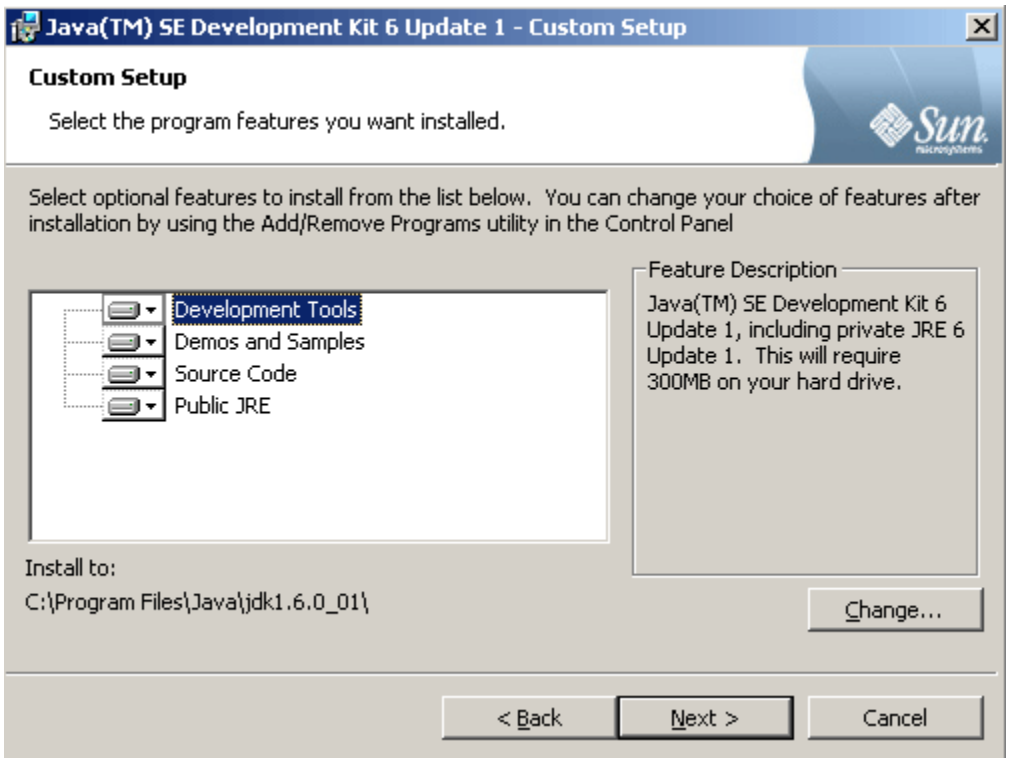




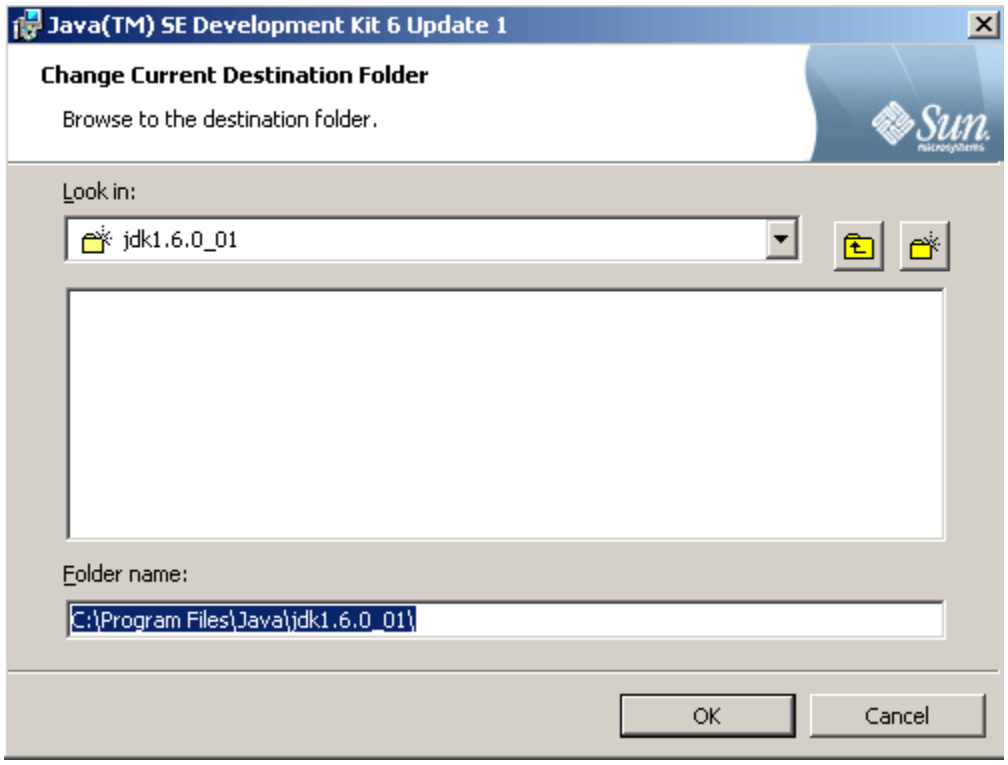
Step 2: Now a "License Agreement" window opens. Just read the agreement and click "Accept" button to accept and go further.



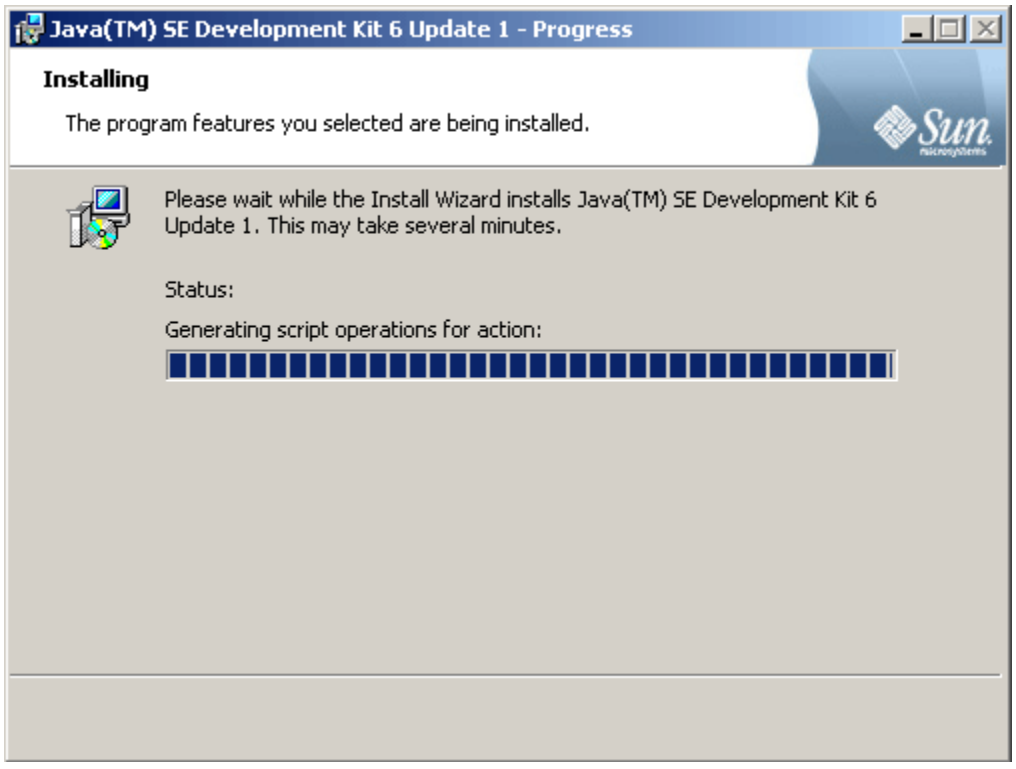
Step 3: Now a "Custom Setup" window opens.

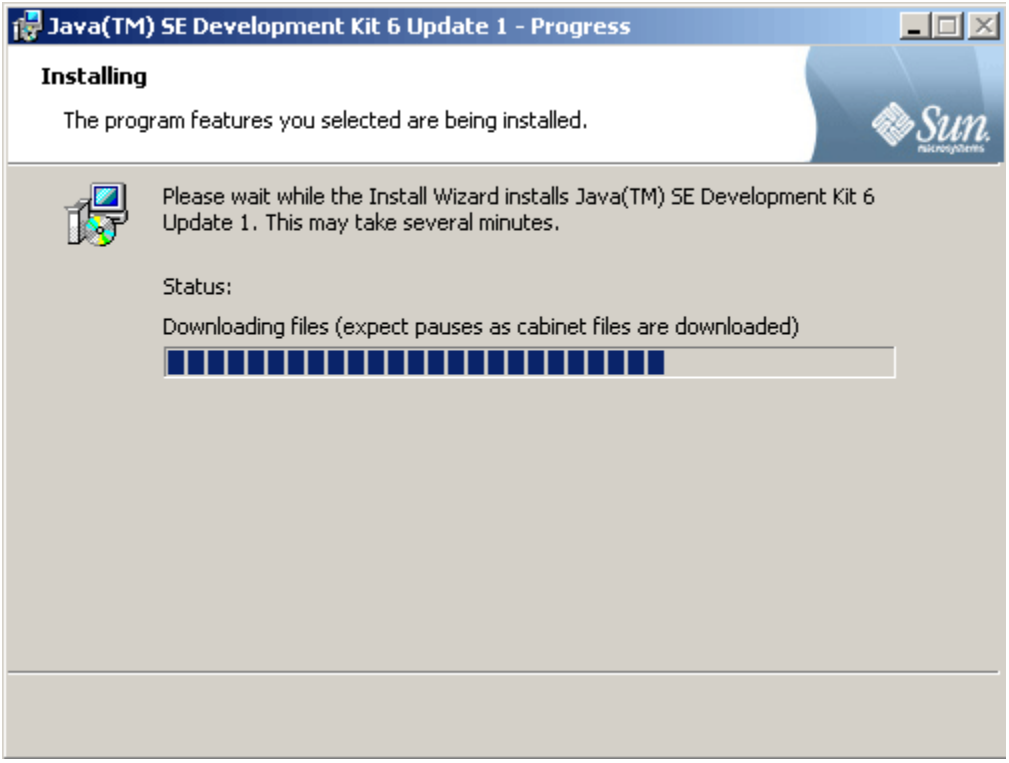


Step 4: Click on "Change" button to choose the installation directory. Here it is "C:\Program Files\ Java\jdk1.6.0_01". Now click on "OK" button.

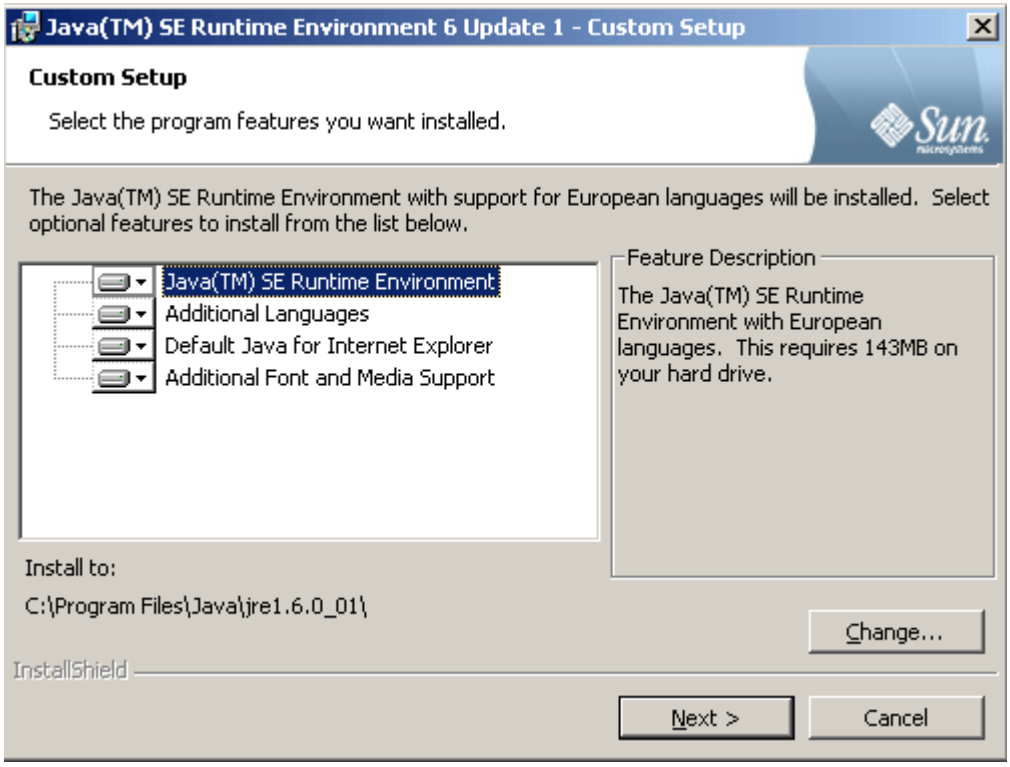


Clicking the "OK" button starts the installation. It is shown in the following figure.

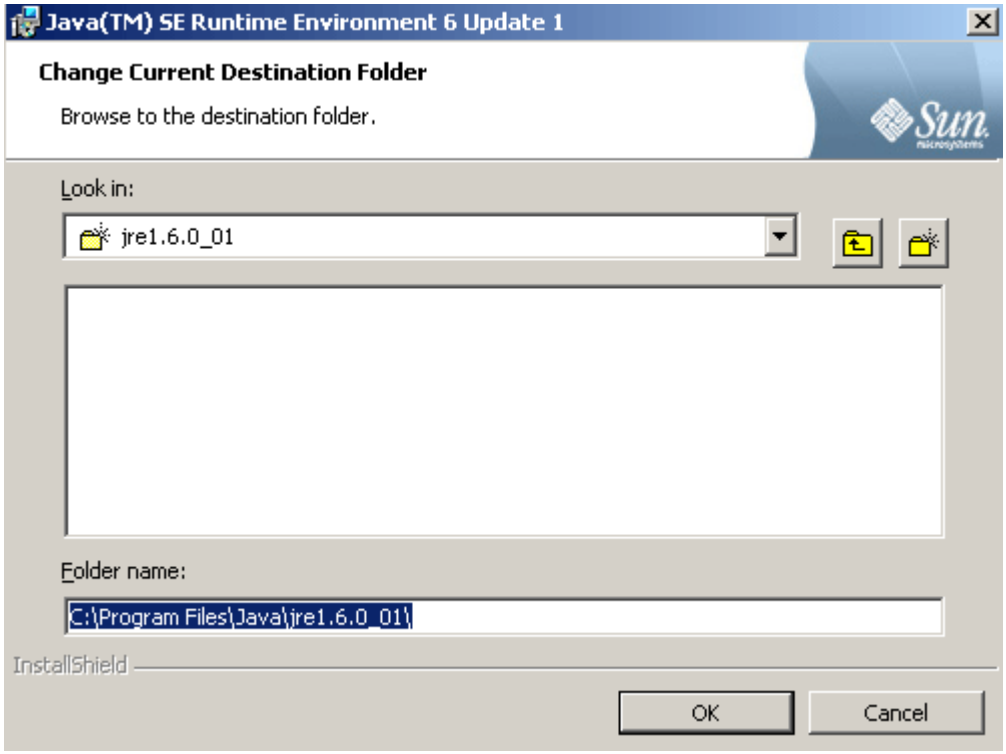




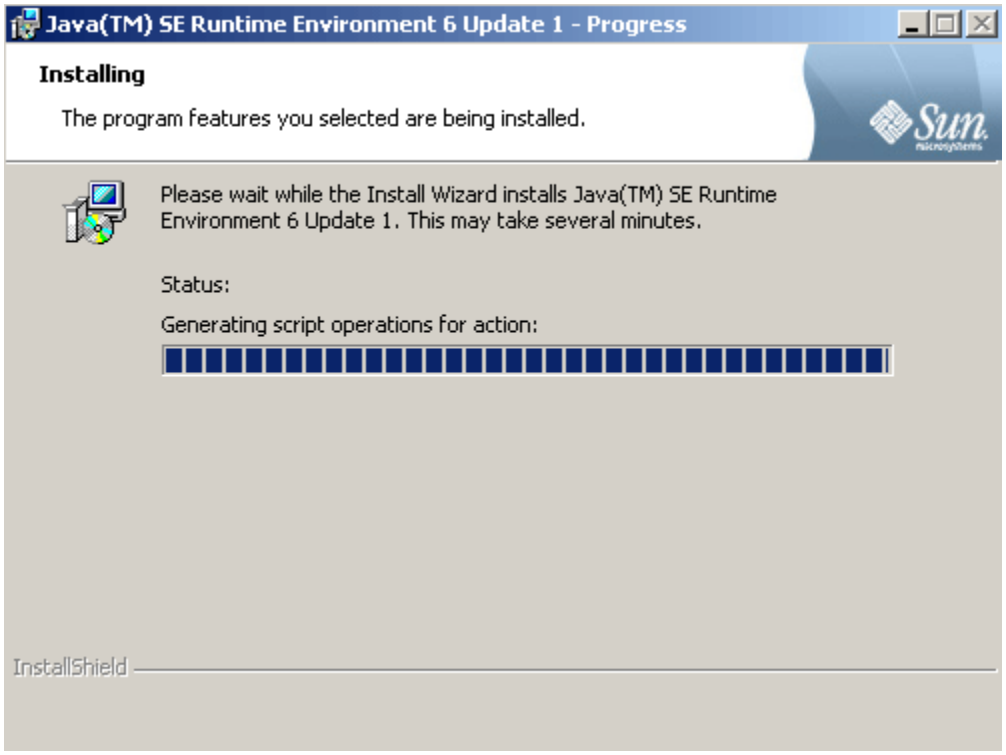
Step 5: Next window asks to install Runtime Environment.



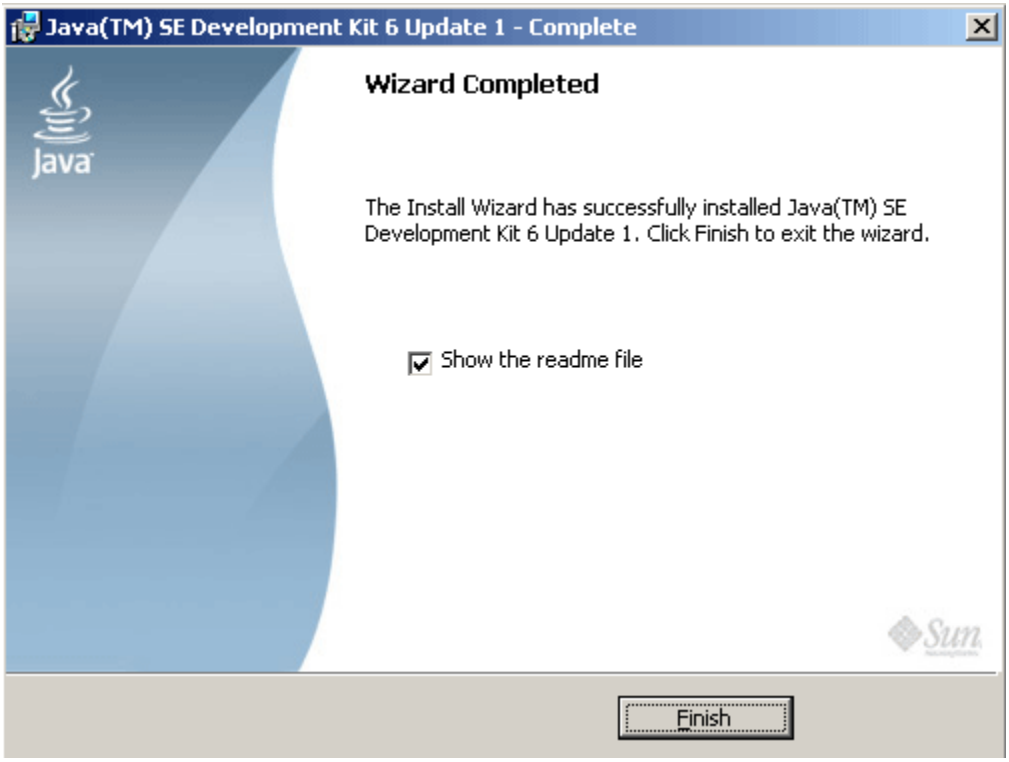
Click the "Change" button to choose the installation directory of Runtime Environment. We prefer not to change it. So click "OK" button.



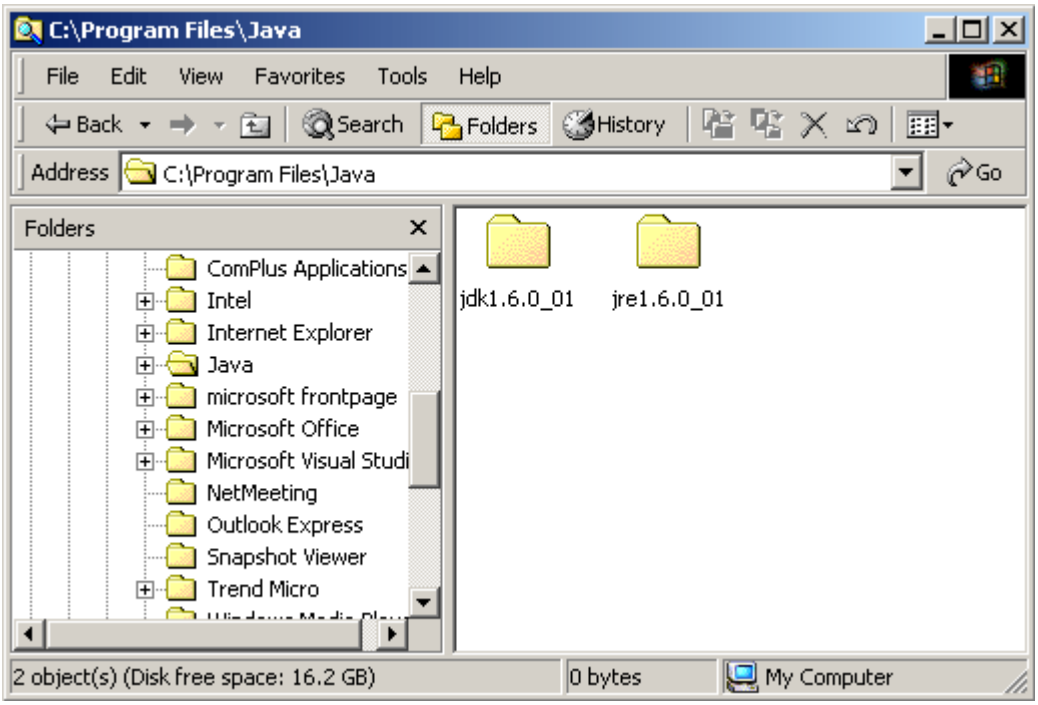
Step 6: Click "OK" button starts the installation.



Step 7: Now "Complete" window appears indicating that installation of jdk 1.6 has completed successfully. Click "Finish" button to exit from the installation process.



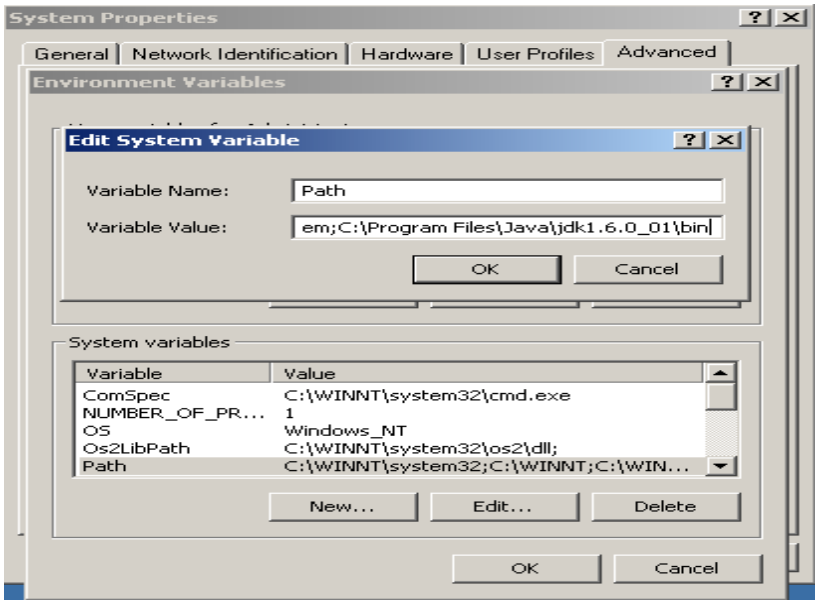
Step 8: The above installation will create two folders "jdk1.6.0_01" and "jre1.6.0_01" in "C:\ Program Files\ java" folder.



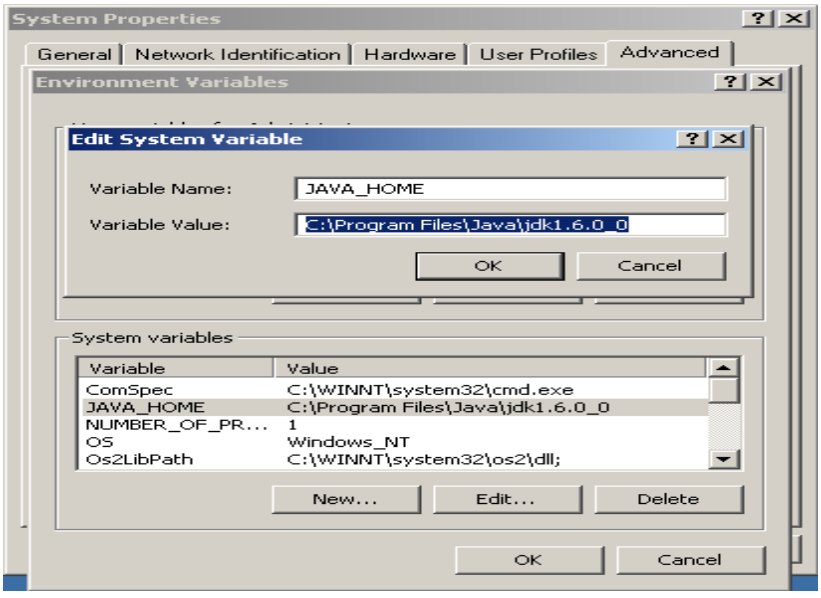
Step 9: To make available Java Compiler and Runtime Environment for compiling and running java programs , we set the system environment variables.

First of all select "My Computer" icon and right click the mouse button. Now click on "system Properties" option. It provides the "System Properties" window , click the 'Advanced' tab. Then Click the "Environment Variables" button. It provides "Environment Variables" window. Now select the path in System variables and click

'Edit' button. The"Edit System Variable" window will open. Add "c:\Program Files\Java\jdk1.6.0_01\bin" to 'variable value' and click 'Ok', 'Ok' and 'Ok' buttons.



Step 10: Now set the JAVA_HOME variable and set its value to " C:\Program Files\Java\jdk1.6.0_01 ". If this variable has not been declared earlier then create a new system variable by clicking on "New" button and give variable name as "JAVA_HOME" and variable value as " C:\Program Files\Java\jdk1.6.0_01 ". Now click "OK". This variable is used by other applications to find jdk installation directory. For example, Tomcat server needs "JAVA_HOME" variable to find the installation directory of jdk.



Step 11: Now this is the final step to check that you have installed jdk successfully and it is working fine. Just go to the command prompt and type javac and hit enter key you will get the screen as shown below :

Now you can create, compile and run java programs.

APACHE TOMCAT INSTALLATION

I'm under the impression that a great number of students are using the campus tomcat installation. The rest of this section should only be of interest to those who wish to run tomcat on their own. I should also mention that there are some development / deployment advantages when working with a local copy of tomcat, coupled with the tomcat plugin for eclipse.

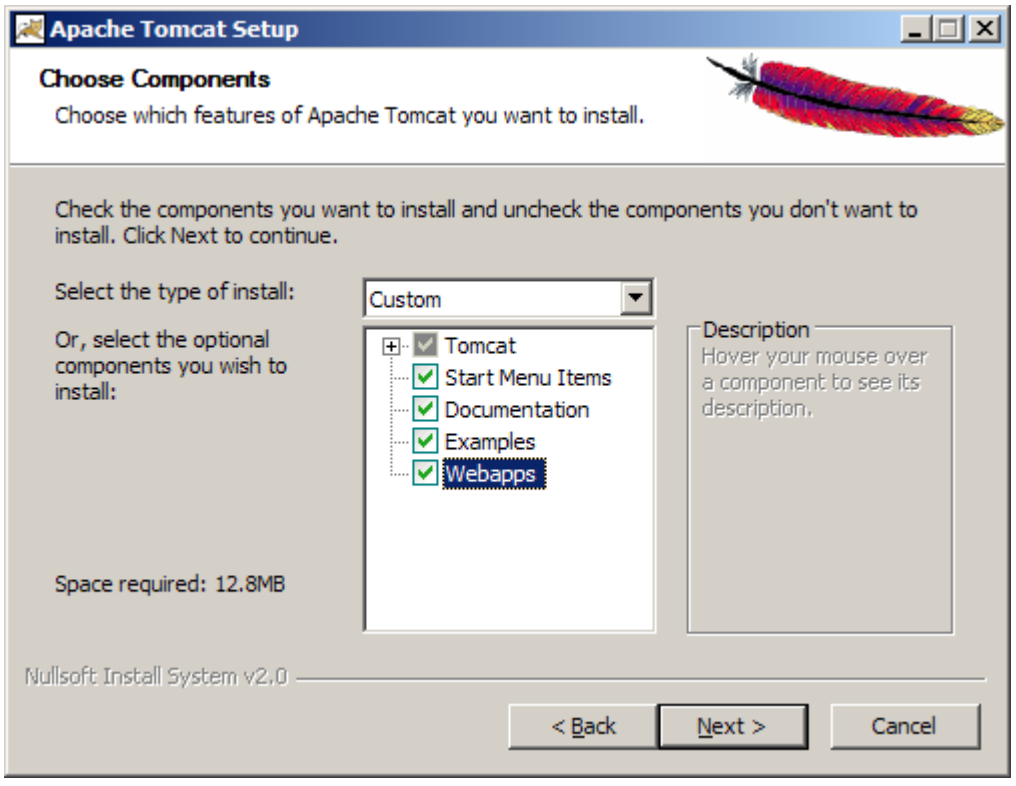
Download the latest version of tomcat from:

http://tomcat.apache.org/

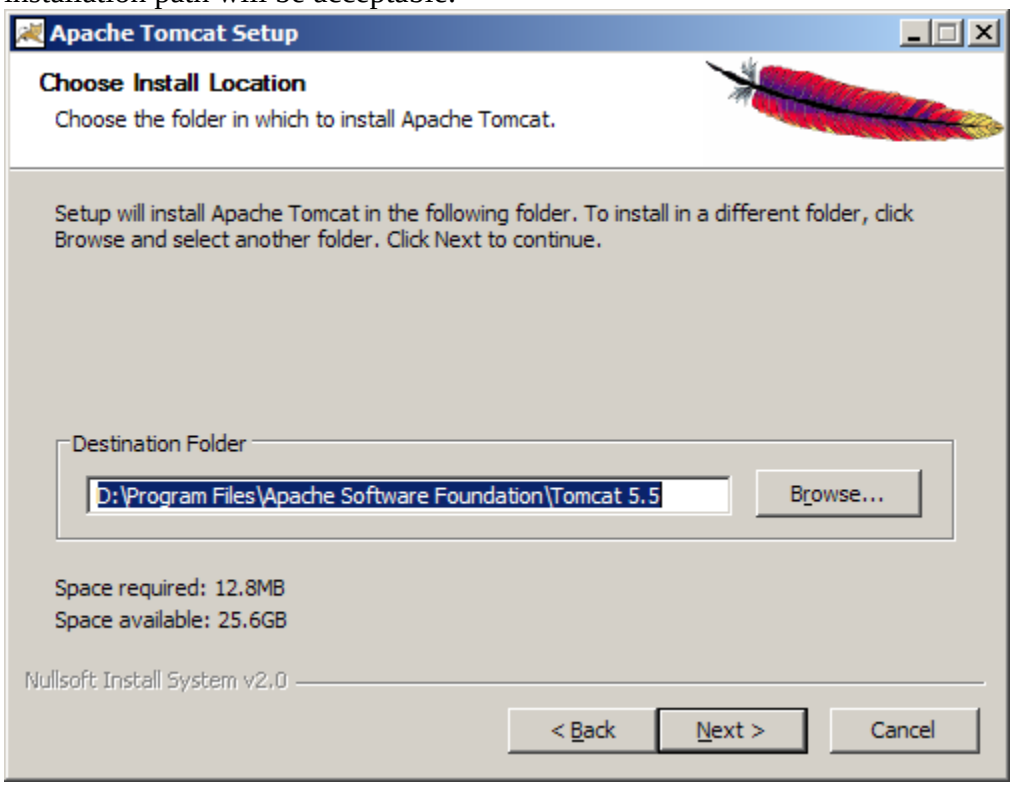
Version 6 of tomcat seems to be in beta at this time. Version 5.x should probably be selected.

I will assume you will download the windows service installer executable for this document. If you wish to have a stand-alone version of the application server, the installation process is very similar (just a little less automated). If you do choose a manual installation, be sure to make a note of where on your file system it lives. You may be visiting that root directory often…

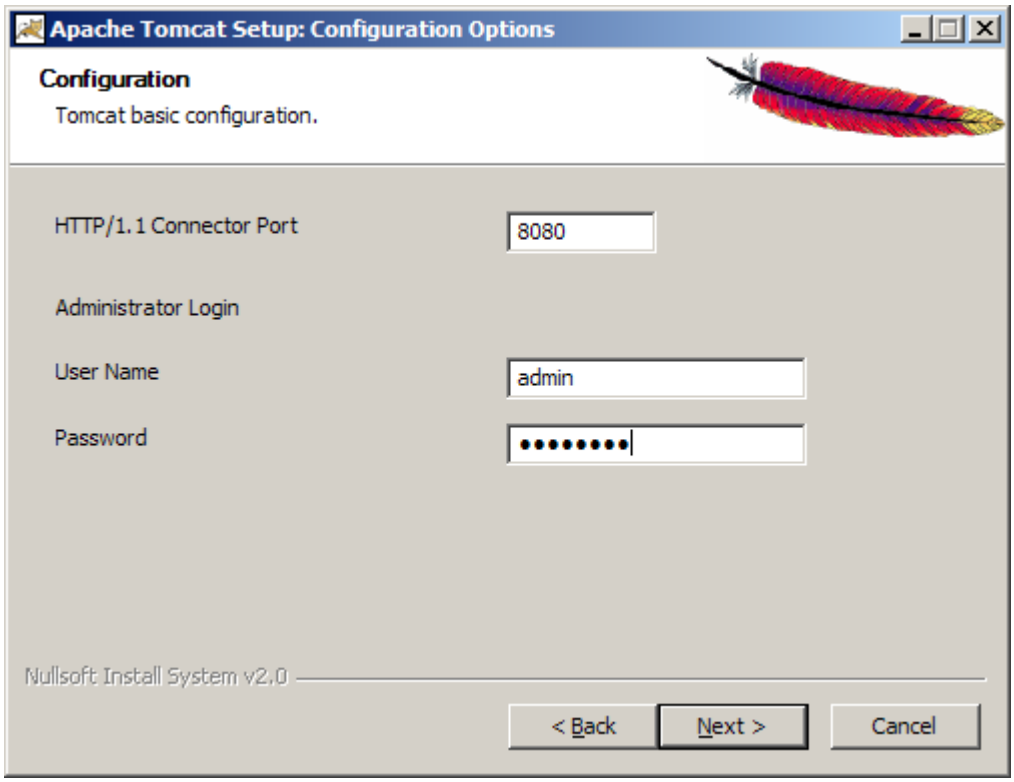
In the initial installation screen, I chose to activate "Examples" and "Webapps" in addition to the defaults.



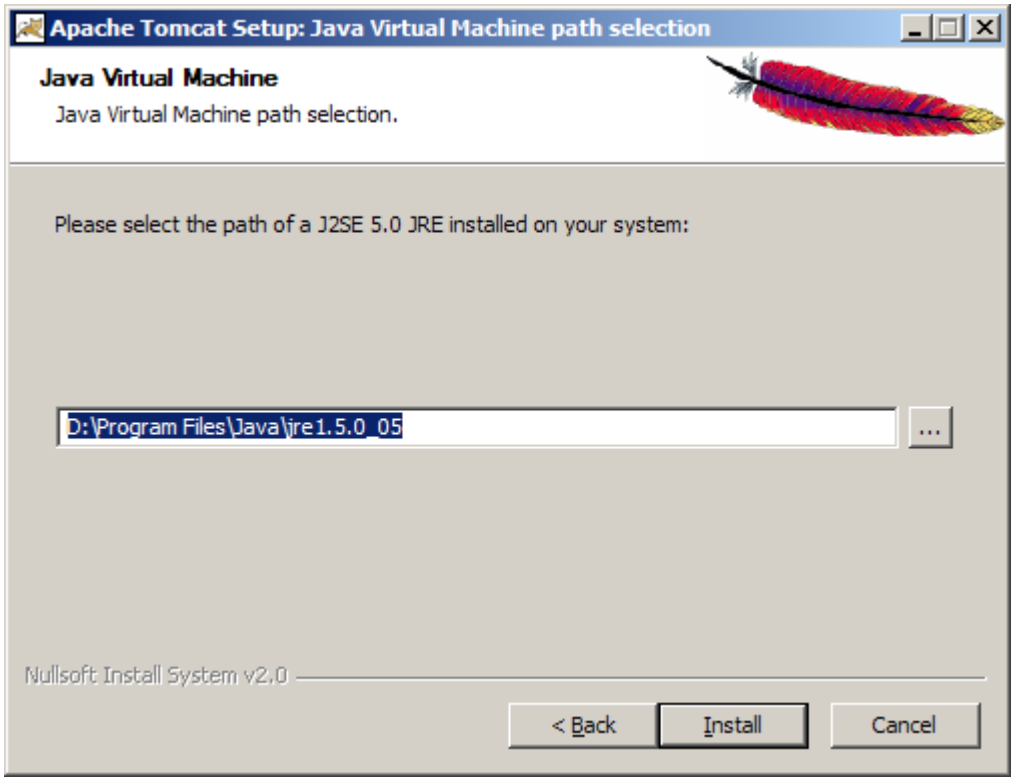
The D: drive is the default on my system, yours may be different. Usually the default installation path will be acceptable.



In the next screen, it is usually safe to leave the port as is. Enter an administration password (don't go forgetting it).



The next screen asks you to locate your installed version of the JRE. If you have not installed it, visit http://java.sun.com and download the latest version for your platform.

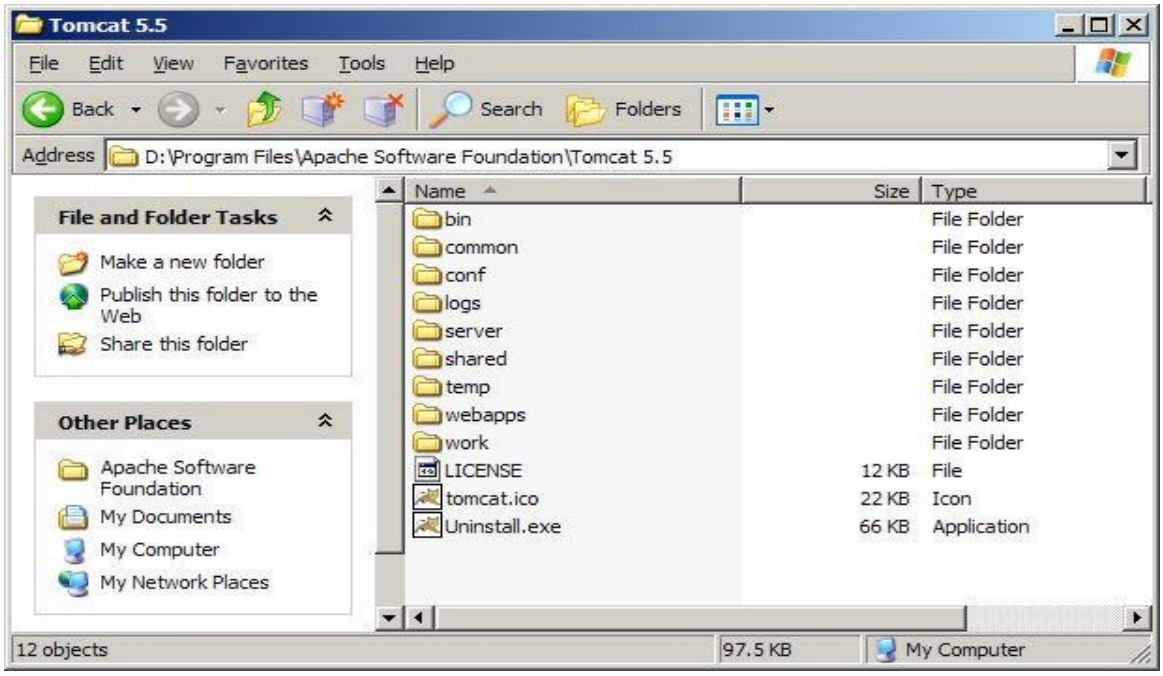


Go ahead and install. The final screen will ask if you wish to start tomcat. Go for it. If everything completed successfully, you should have a new icon in your system tray indicating that tomcat is running. Double-clicking on it will bring up a configuration menu, check it out.

Open a web browser and navigate to http://localhost:8080/ you should be looking at the main tomcat administration / configuration panel. Some examples and administrative functions are interesting, feel free to poke around.

On the file system, navigate to your specific installation directory (in my case it is D:\Program Files\Apache Software Foundation\Tomcat 5.5). For the rest of this document I will refer to this installation directory as PREFIX

Note the directory layout:



You'll want to note the "bin" directory, as well as the "webapps" dir. The bin directory contains several scripts and batch files, notably startup and shutdown scripts. If you're running this as a service then you can more or less ignore the bin directory.

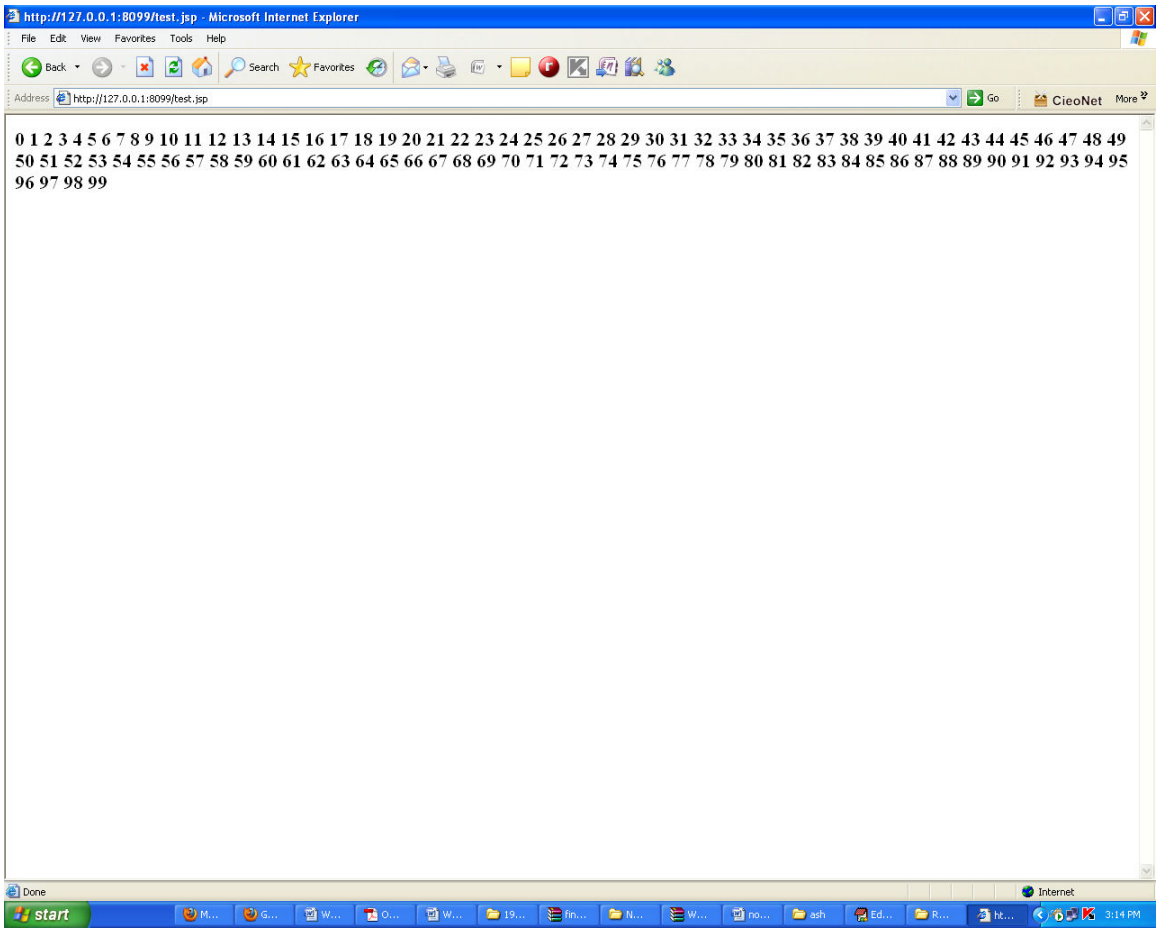
The webapps directory is where all of our magic will happen. For our first jsp, navigate to PREFIX/webapps/ROOT and create a new text document "test.jsp". Windows may complain about not understanding the file extension. Open this file with wordpad / textpad / editor of choice.

Code for test.jsp follows:

```
<%@ page language="Java" %>      <!-- this is the directive -->

<html>
<body>
<%
        // the for loop starts here, we'll close this part of the scriptlet to display our results

        for (int i=0; i < 100; i++) {
%>
<b>

<%= i %>
</b>

<% } %> <!-- do not forget to close the for loop… -->

</body>
</html>
```

Now save this test.jsp in the ROOT directory, and point your browser to http://localhost:8080/test.jsp

You should see the numbers 0-99 displayed in bold-faced font.



Some notes about jsps: Any jsp is compiled by the application server into a servlet, and then a java class file, the first time it is requested. Asp, on the other hand, is interpreted every time and no compilation is performed. It is sometimes desirable for developers to edit their jsps and manually delete the servlets, forcing the application server to recompile the fresh code, though most of the time tomcat will recognize that the jsp is new and take care of the recompilation automatically.