WEB TECHNOLOGIES

UNIT-I: HTML Common tags- List, Tables, images, forms, Frames; Cascading Style sheets;

UNIT-II: Introduction to Java Scripts, Objects in Java Script, Dynamic HTML with Java Script

UNIT-III: XML: Document type definition, XML Schemas, Document Object model, Presenting XML, Using XML Processors: DOM and SAX.

UNIT-IV:

Java Beans: Introduction to Java Beans, Advantages of Java Beans, BDK Introspection, Using Bound properties, Bean Info Interface, Constrained properties , Persistence, Customizes, Java Beans API, Introduction to EJB's

UNIT-V:

Web Servers and Servlets: Tomcat web server, Introduction to Servelets: Lifecycle of a Serverlet, JSDK, The Servelet API, The javax.servelet Package, Reading Servelet parameters, Reading Initialization parameters. The javax.servelet HTTP package, Handling Http Request & Responses, Using Cookies-Session Tracking, Security Issues,

UNIT-VI:

Introduction to JSP: The Problem with Servelet. The Anatomy of a JSP Page, JSP Processing. JSP Application Design with MVC Setting Up and JSP Environment: Installing the Java Software Development Kit, Tomcat Server & Testing Tomcat

UNIT-VII:

JSP Application Development: Generating Dynamic Content, Using Scripting Elements Implicit JSP Objects, Conditional Processing – Displaying Values Using an Expression to Set an Attribute, Declaring Variables and Methods Error Handling and Debugging Sharing Data Between JSP pages, Requests, and Users Passing Control and Date between Pages – Sharing Session and Application Data – Memory Usage Considerations

UNIT VIII:

Database Access : Database Programming using JDBC, Studying Javax.sql.* package,Accessing a Database from a JSP Page, Application – Specific Database Actions,Deploying JAVA Beans in a JSP Page, Introduction to struts framework..

TEXT BOOKS:

1. Web Programming, building internet applications, Chris Bates 2nd edition, WILEY Dreamtech (UNIT s 1,2 ,3)
2. The complete Reference Java 2 Fifth Edition by Patrick Naughton and Herbert Schildt. TMH (Chapters: 25) (UNIT 4)
3. Java Server Pages –Hans Bergsten, SPD O'Reilly (UNITs 5,6,7,8)

REFERENCE BOOKS:

1. Programming world wide web-Sebesta,Pearson
2. Core SERVLETS ANDJAVASERVER PAGES VOLUME 1: CORE TECHNOLOGIES By Marty Hall and Larry Brown Pearson
3. Internet and World Wide Web – How to program by Dietel and Nieto PHI/Pearson Education Asia.
4. Jakarta Struts Cookbook , Bill Siggelkow, S P D O'Reilly for chap 8.
5. Murach's beginning JAVA JDK 5, Murach, SPD
6. An Introduction to web Design and Programming –Wang-Thomson
7. Web Applications Technologies Concepts-Knuckles,John Wiley
8. Programming world wide web-Sebesta,Pearson
9. Web Warrior Guide to Web Programmming-Bai/Ekedaw-Thomas
10. Beginning Web Programming-Jon Duckett WROX. ,

UNIT I : INTRODUCTION TO HTML(Hypertext markup language)

CONTENTS

- Introduction
- Structure of html
- Basic tags
  - Head tag
  - Title tag
  - Body tag with attributes
  - Formatting tags
  - Heading tag
- List tag with an example
- Table tag with an example
- Images tag with an example
- Frame tag with an example
- Forms
- Cascading style sheets

INTRODUCTION TO HTML

HTML, or HyperText Markup Language is designed to specify the logical organisation of a document, with important hypertext extensions. It is *not* designed to be the language of a word processor such as Word. HTML allows you to mark selections of text as titles or paragraphs, and then leaves the interpretation of these marked *elements* up to the browser. For example one browser may indent the beginning of a paragraph, while another may only leave a blank line.

HTML instructions divide the text of a document into blocks called *elements*. These can be divided into two broad categories -- those that define how the BODY of the document is to be displayed by the browser, and those that define information `about' the document, such as the title or relationships to other documents.

The detailed rules for HTML (the names of the tags/elements, how they can be used) are defined using another language known as the standard generalized markup language, or SGML. SGML is wickedly difficult, and was designed for massive document collections. Fortunately, HTML is much simpler!

However, SGML has useful features that HTML lacks. For this reason, markup language and software experts have developed a new language, called XML (the *eXtensible markup language*) which has most of the most useful features of HTML and SGML.

History of HTML

HTML was originally developed by Tim Berners-Lee while at CERN, and popularized by the Mosaic browser developed at NCSA. During the course of the 1990s it has blossomed with the explosive growth of the Web. During this time, HTML has been extended in a number of ways. The Web depends on Web page authors and vendors sharing the same conventions for HTML. This has motivated joint work on specifications for HTML.

HTML 2.0 (November 1995) was developed under the aegis of the Internet Engineering Task Force (IETF) to codify common practice in late 1994.HTML 3.0 (1995) proposed much richer versions of HTML.

Achieving interoperability lowers costs to content providers since they must develop only one version of a document. If the effort is not made, there is much greater risk that the Web will devolve into a proprietary world of incompatible formats, ultimately reducing the Web's commercial potential for all participants.

Each version of HTML has attempted to reflect greater consensus among industry players so that the investment made by content providers will not be wasted and that their documents will not become unreadable in a short period of time.

HTML has been developed with the vision that all manner of devices should be able to use information on the Web: PCs with graphics displays of varying resolution and color depths, cellular telephones, hand held devices, devices for speech for output and input, computers with high or low bandwidth, and so on.

Advantages of HTML:

1. First advantage it is widely used.
2. Every browser supports HTML language.
3. Easy to learn and use.
4. It is by default in every windows so you don't need to purchase extra software.

Disadvantages of HTML:

1. It can create only static and plain pages so if we need dynamic pages then HTML is not useful.
2. Need to write lot of code for making simple webpage.
3. Security features are not good in HTML.
4. If we need to write long code for making a webpage then it produces some complexity.

Important points

- Tags are delimited by angled brackets.
- They are not case sensitive i.e., <head>, <HEAD> and <Head> is equivalent.
- If a browser not understand a tag it will usually ignore it.
- Some characters have to be replaced in the text by escape sequences.
- White spaces, tabs and newlines are ignored by the browser.

Structure of an HTML document:

All HTML documents follow the same basic structure. They have the root tag as <html>, which contains <head> tag and <body> tag. The head tag is used for control information by the browser and the body tag contains the actual user information that is to be displayed on the screen. The basic document is shown below.

```
<html>
<head>
<title> Basic HTML document </title>
</head>
<body>
<h1> Welcome to the world of Web Technologies</h1>
<p> A sample html program </p>
</body>
</html>
```

Besides head and body tag, there are some other tags like title, which is a sub tag of head, that displays the information in the title bar of the browser. <h1> is used to display the line in its own format i.e., bold with some big font size. <p> is used to write the content in the form of paragraph.
Comments in HTML documents start with <! and end with >. Each comment can contain as many lines of text as you like. If comment is having more lines, then each line must start and end with -- and must not contain -- within its body.

<! -- this is a comment line - -
-- which can have more lines - ->

Basic HTML tags

Body tag :

Body tag contain some attributes such as bgcolor, background etc. bgcolor is used for background color, which takes background color name or hexadecimal number and #FFFFFF and background attribute will take the path of the image which you can place as the background image in the browser.
<body bgcolor=”#F2F3F4” background= “c:\btech\imag1.gif”>

Paragraph tag:

Most text is part of a paragraph of information. Each paragraph is aligned to the left, right or center of the page by using an attribute called as align.

<p align=”left” | “right” | “center”>

Heading tag:

HTML is having six levels of heading that are commonly used. The largest heading tag is <h1> . The different levels of heading tag besides <h1> are <h2>, <h3>, <h4>, <h5> and <h6>. These heading tags also contain attribute called as align.
<h1 align=”left” | “right” | “center”> . . . . <h2>

hr tag:

This tag places a horizontal line across the system. These lines are used to break the page. This tag also contains attribute i.e., width which draws the horizontal line with the screen size of the browser. This tag does not require an end tag.
<hr width=”50%”>.

font tag:

This sets font size, color and relative values for a particular text.
<font size=”10” color=”#f1f2f3”>

bold tag:

This tag is used for implement bold effect on the text
 …….
Italic tag:

This implements italic effects on the text.
<i>…….</i>
strong tag:

This tag is used to always emphasized the text
……….
sub and sup tag:
These tags are used for subscript and superscript effects on the text.
_{……….}
^{………..}
Break tag:
This tag is used to the break the line and start from the next line.

& < >   "

These are character escape sequence which are required if you want to display characters that HTML uses as control sequences.
Example: < can be represented as <.
Anchor tag:

This tag is used to link two HTML pages, this is represented by <a>

 some text
href is an attribute which is used for giving the path of a file which you want to link.

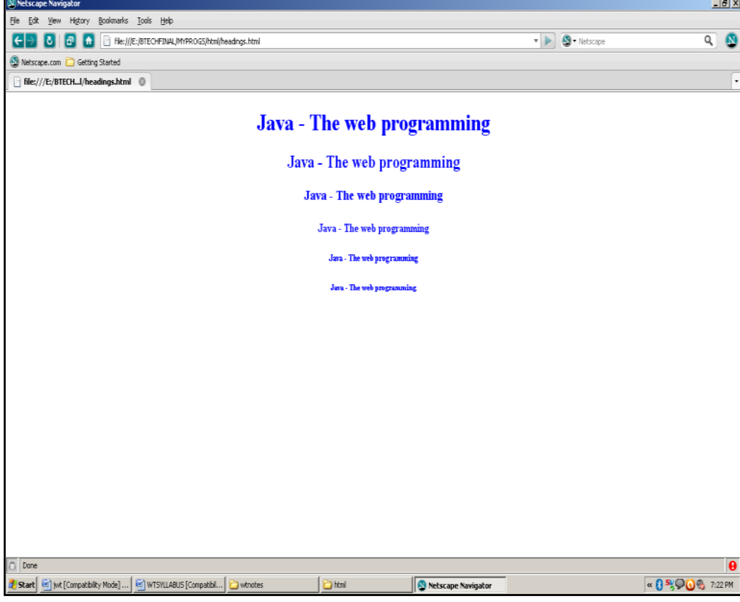
Example 1: HTML code to implement common tags.

mypage.html

```
<html>
<head> <! -- This page implements common html tags -->
<title> My Home page </title>
</head>
<body >
<h1 align="center"> VIVEKANANDA INSTITUTE OF TECHNOLOGICAL SCIENCES </h1>
<h2 align="center"> Karimnagar</h2>
</body>
</html>
```

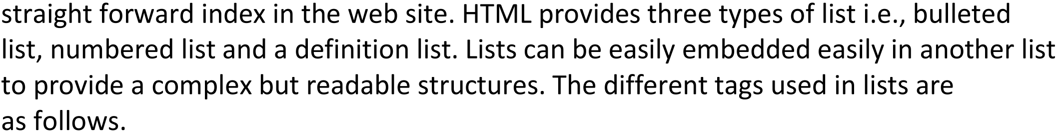
<! -- using heading tags -->

```
<html>
<body  text=blue>
<center>
<h1> Java - The web programming
<h2> Java - The web programming
<h3> Java - The web programming
<h4> Java - The web programming
<h5> Java - The web programming
<h6> Java - The web programming
</center>
</body>
</html>
```



Lists:

One of the most effective ways of structuring a web site is to use lists. Lists provides straight forward index in the web site. HTML provides three types of list i.e., bulleted list, numbered list and a definition list. Lists can be easily embedded easily in another list to provide a complex but readable structures. The different tags used in lists are as follows.



 …..

The ordered(numbered) and unordered(bulleted) lists are each made up of sets of list items. This tag is used to write list items

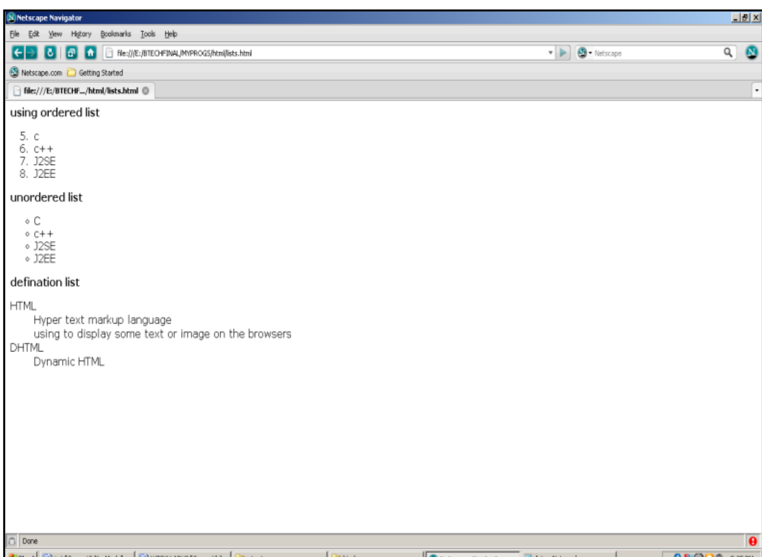
**<ul type="disc" | "square" | "circle" > ….. **

This tag is used for basic unordered list which uses a bullet in front of each tag, every thing between the tag is encapsulated within <li> tags.

**<ol type="1" | "a" | "I" start="n">….. **

This tag is used for unordered list which uses a number in front of each list item or it uses any element which is mentioned in the type attribute of the <ol> tag, start attribute is used for indicating the starting number of the list.

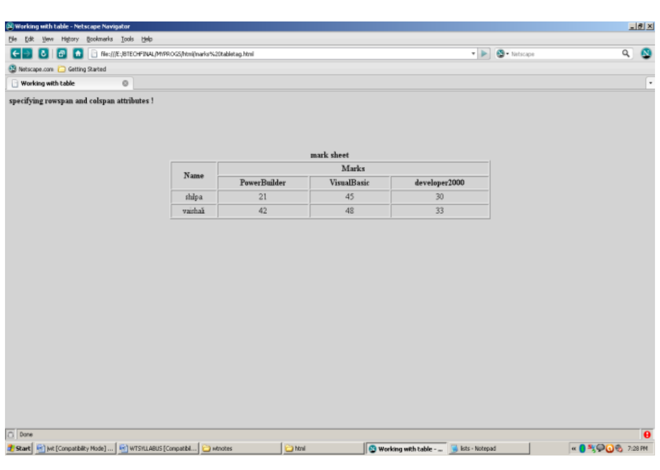
<dl>….. </dl>
This tag is used for the third category i.e., definition list, where numbers or bullet is not used in front of the list item, instead it uses definition for the items.
<dt>…..</dt>
This is a sub tag of the <dl> tag called as definition term, which is used for marking the items whose definition is provided in the next data definition.
<dd> ….</dd>
This is a sub tag of the <dd> tag, definition of the terms are enclosed within these tags. The definition may include any text or block.




```
<! -- using lists -->
<html>
<body>
<font face=Verdana size=3>
<b> using ordered list </b>
<ol type=1 start=5>
<li> c
<li> c++
<li> J2SE
<li> J2EE
</ol>
<b> unordered list </b>
<ul type="circle">
<li> C
<li> c++
<li> J2SE
<li> J2EE
</ul>
<b> definition list </b>
<dl>
<dt> HTML
<dd> Hyper text markup language
<dd> using to display some text or image on the browsers
<dt> DHTML
<dd>  Dynamic HTML
</dl>
</fonts>
</body>
</html>
```

Tables:
Table is one of the most useful HTML constructs. Tables are find all over the web application. The main use of table is that they are used to structure the pieces of information and to structure the whole web page. Below are some of the tags used in table.
<table align=”center” | “left” | “right” border=”n” width=”n%” cellpadding=”n” cellspacing=”n”>………………</table>

Every thing that we write between these two tags will be within a table. The attributes of the table will control in formatting of the table. Cell padding determines how much space there is between the contents of a cell and its border, cell spacing sets the amount of white space between cells. Width attribute sets the amount of screen that table will use.
<tr> ….. </tr>
This is the sub tag of <table> tag, each row of the table has to be delimited by these tags.
<th>……</th>
This is again a sub tag of the <tr> tag. This tag is used to show the table heading .
<td>…..</td>
This tag is used to give the content of the table.




```
<html>
<head>
<title> Working with table </title>
</head>
<body bgcolor=lightgrey>
<b> specifying rowspan and colspan attributes ! </b>
<br><br><br><br>
<center>
<table border=1 width=50% align=center>
        <tr>
                <th rowspan=2>Name
                <th colspan=3>Marks
        </tr>
        <tr>
                <th>PowerBuilder
                <th>VisualBasic
                <th>developer2000
        </tr>
        <tr align=center>
                <td>shilpa
                <td>21
                <td>45
                <td>30
        </tr>
        <tr align=center>
                <td>vaishali
                <td>42
                <td>48
                <td>33
        </tr>


<caption align=bottom> <b> <br>
mark sheet </b>
</caption
</table>
</center>
</body>  </html>
```

Color and Image:
Color can be used for background, elements and links. To change the color of links or of the page background hexadecimal values are placed in the <body> tag.
<body bgcolor = “#nnnnnn” text = “#nnnnnn” link= “#nnnnnn” vlink= “#nnnnnn” alink = “#nnnnnn”>
The vlink attribute sets the color of links visited recently, alink the color of a currently active link. The six figure hexadecimal values must be enclosed in double quotes and preceded by a hash(#).
Images are one of the aspect of web pages. Loading of images is a slow process, and if too many images are used, then download time becomes intolerable. Browsers display a limited range of image types.
<body background = “URL”>
This tag will set a background image present in the URL.
Another tag that displays the image in the web page, which appears in the body of the text rather than on the whole page is given below
<img src=”URL” height=”n” width=”n” align = “top” | “center” | “bottom” >

Frames:
Frames provide a pleasing interface which makes your web site easy to navigate. When we talk about frames actually we are referring to frameset, which is a special type of web page. The frameset contains a set of references to HTML files, each of which is displayed inside a separate frame. There are two tags related to frames i.e., frameset and frame
<frameset cols=” % , %” | rows=” % , %”>……..</frameset>
<frame name=”name” src=”filename” scrolling =” yes” | “no” frameborder =”0”|”1”>

Forms:
Forms are the best way of adding interactivity of element in a web page. They are usually used to let the user to send information back to the server but can also be used to simplify navigation on complex web sites. The tags that use to implement forms are as follows.
<forms action=”URL” method = “post” | “get”>…….</form>
When get is used, the data is included as part of the URL. The post method encodes the data within the body of the message. Post can be used to send large amount of data, and it is more secure than get. The tags used inside the form tag are:
<input type = “text” | “password” | “checkbox” | “radio” | “submit” name=”string” value=”string” size=”n”>
In the above tag, the attribute type is used to implement text, password, checkbox, radio and submit button.
Text: It is used to input the characters of the size n and if the value is given than it is used as a default value. It uses single line of text. Each component can be given a separate name using the name attribute.
Password: It works exactly as text, but the content is not displayed to the screen, instead an * is used.
Radio: This creates a radio button. They are always grouped together with a same name but different values.
Checkbox: It provides a simple checkbox, where all the values can be selected unlike radio button.

Submit: This creates a button which displays the value attribute as its text. It is used to send the data to the server.

<select name=”string”>…..</select>

This tag helps to have a list of item from which a user can choose. The name of the particular select tag and the name of the chosen option are returned.

<option value=”string” selected>……</option>

The select statement will have several options from which the user can choose. The values will be displayed as the user moves through the list and the chosen one returned to the server.

<textarea name=”string” rows=”n” cols=”n”>…….</textarea>

This creates a free format of plain text into which the user can enter anything they like. The area will be sized at rows by cols but supports automatic scrolling.

Examples

```
<HMTL><HEAD>
<SCRIPT Language='JavaScript'>
function Chk(f1)
{
 if(f1.Check.checked)
    alert(" The Checkbox just got checked ");
else
    alert("not checked");
    f1.Radio[1].checked=true;
    f1.Radio[0].checked=false;
    alert(" The Radio button just got checked ");

}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
Client Name :
<Input Type=Text Name="Text" Value=""><BR><BR>
Client Address:
<Input Type=Text Name="Text1" Value=""> <BR><BR>
Client E-mail Address :<Input Type=Text Name="Text2" Value=""><BR><BR>
<Input Type="radio" Name="Radio" Value=""> Male
<Input Type="radio" Name="Radio" Value=""> Female<BR><BR>
<Input Type="CheckBox" Name="Check" Value=""> Employed <BR><BR>
<Input Type="Button" Name="Bt" Value="Set Element Array Value" onClick="Chk(this.form)">
</FORM></BODY></HTML>
```

CASCADING STYLESHEETS

One of the most important aspects of HTML is the capability to separate presentation and content. A style is simply a set of formatting instructions that can be applied to a piece of text. There are three mechanisms by which we can apply styles to our HTML documents.

- Style can be defined within the basic HTML tag.
- Style can be defined in the <head> tag
- Styles can be defined in external files called stylesheets which can then be used in any document by including the stylesheet via a URL.

A style has two parts: a selector and a set of declarations. The selector is used to create a link between the rule and the HTML tag. The declaration has two parts: a property and a value. Declarations must be separated using colons and terminated using semicolons.

Selector{property: value; property: value …….}

Properties and values in styles:

Font Attributes	**Values**
Font-family	Comma is delimiter, sequence of fonts like cursive ,sans etc
Font-style	Normal , italic oblique
Font-weight	Normal,bold,bolder,lighter,or one of these numerical values(100 t0 900)
Font-size	It is absolute size (xx-small,x-small,small,medium,large,x-large,xx-large),relative size(larger,smaller),a number(pixels)

Color and background Attributes	**Values**
Color	Sets an element text color
Background-color	Used to set back color
Background-image	Set background image

Text Attributes	**Values**
Text-decoration	None,underline,overline,line-through,blink
Vertical-align	top,bottom.middle,text-top,text-bottom
Text-transform	Capitalize,uppercase,lowercase
Text-align	Left,right,center,justify

Measurement units

Unitname	**abbreviation**	**Explanation**
Em	Em	Height of the font
Pica	pc	1 pica is 12 points
Point	pt	1/72 of inch
pixel	px	One dot on screen
millimeter	mm	Printing unit
Centimeter	cm	Printing unit
inch	in	Printing unit

Marging related Arrtibutes	Values
Margin-top	Percentage or length
Margin-bottom	Length or percentage
Margin-left	Length or percentage
Margin-right	Length or percentage

HTML code representing cascading style sheet

```
<html>
<head>
<title>My Web Page</title>
<style type="text/css">
h1{font-family:mssanserif;font-size:30;font-style:italic;fontweight:
bold;color:red;background-color:blue;border:thin groove}
.m{border-width:thick;border-color:red;border-style:dashed}
.mid{font-family:BankGothicLtBT;text-decoration:link;texttransformation:uppercase;text-indentation:60%}
</style>
</head>
<body class="m">
<h1> VITS Engineering College</h1>
<p class="mid">Jawaharlal Technological University Hyderabad</p>
</div>
</body>
</html>
```

UNIT-II: JAVA SCRIPT & DHTML

CONTENTS

- Introduction to Java Scripts
  - Variable , operators
  - Conditional statements ,loops
  - Functions
  - Events
- Objects in Java Script
  - Window
  - Navigator
  - Document
  - form
  - Date
  - String
  - arrays
- Dynamic HTML with Java Script

UNIT II

Introduction to JavaScript

A number of technologies are present that develops the static web page, but we require a language that is dynamic in nature to develop web pages a client side. Dynamic HTML is a combination of content formatted using HTML, cascading stylesheets, a scripting language and DOM.

JavaScript originates from a language called LiveScript. The idea was to find a language which can be used at client side, but not complicated as Java. JavaScript is a simple language which is only suitable for simple tasks.

Benefits of JavaScript

Following are some of the benefits that JavaScript language possess to make the web site dynamic.

- It is widely supported in browser
- It gives easy access to document object and can manipulate most of them.
- JavaScript can give interesting animations with many multimedia datatypes.
- Special plug-in are not required to use JavaScript
- JavaScript is secure language
- JavaScript code resembles the code of C language, The syntax of both the language is
- very close to each other. The set of tokens and constructs are same in both the language.

A Sample JavaScript program

```
<html>
<head><title>java script program</title>
<script languague="javascript">
function popup()
{
var major=parseInt(navigator.appVersion);
var minor=parseInt(navigator.appVersion);
var agent=navigator.userAgent.toLowerCase();
document.write(agent+" "+major);
window.alert(agent+" "+major);
}
function farewell()
{
window.alert("Farewell and thanks for visiting");
}
</script>
</head>
<body onLoad="popup()" onUnload="farewell()">
</body>
</html>
```

- JavaScript program contains variables, objects and functions.
- Each line is terminated by a semicolon. Blocks of code must be surrounded by curly brackets.
- Functions have parameters which are passed inside parenthesis
- Variables are declared using the keyword var.
- Script does not require main function and exit condition.

JavaScript program that shows the use of variables, datatypes

```
<html>
<head>
<title> My Sample JavaScript program</title>
<script language="javascript">
function disp()
{
var rno,sname,br,pr;
rno=prompt("Enter your registration number");
sname=prompt("Enter your Name");
br=prompt("Enter your branch Name");
pr=prompt("Enter the percentage");
document.writeln("<h2> Your Registration No. is :</h2>"+ rno.toUpperCase());
document.writeln("<h2> Your Name is :</h2>"+sname.toUpperCase());
document.writeln("<h2> Your Branch Name is :</h2>"+br.toUpperCase());
document.writeln("<h2> Your Overall Percentage is :</h2>"+pr);
document.close();
}
</script>
</head>
<body onLoad="disp()">
</body>
</html>
```

JavaScript program showing the using of constructs

```
<html>
<head> <title> Factorial</title> </head.
<body>
<script language="javascript">
function fact(n)
{
var i,f=1;
for(i=1;i<=n;i++)
{
f=f*i;
}
return(f);
}
var x,n,f;
x=prompt("Enter the number");
f=fact(x);
document.writeln("Factorial of "+x+" is "+f);
document.close();
</script>
</body>
</html>
```

Variables

Variables are like storage units. You can create variables to hold values. It is ideal to name a variable something that is logical, so that you'll remember what you are using it for. For example, if you were writing a program to divide 2

numbers, it could be confusing if you called your variables numberOne, numberTwo, numberThree because you may forget which one is the divisor, which one is the dividend, and which one is the quotient. A more logical approach would be to name them just that: divisor, dividend, quotient.

It is important to know the proper syntax to which variables must conform:

- They must start with a letter or underscore ("_")
- Subsequent characters can also be digits (0-9) or letters (A-Z and/or a-z). Remember, JavaScript is case-sensitive. (That means that MyVariable and myVariable are two different names to JavaScript, because they have different capitalization.)

Some examples of legal names are Number_hits, temp99, and _name.

When you declare a variable by assignment outside of a function, it is called a global variable, because it is available everywhere in the current document. When you declare a variable within a function, it is called a local variable, because it is available only within the function. Using var is optional, but you need to use it if you have a variable that has been declared global and you want to re-declare it as a local variable inside a function.

Variables can store all kinds of data (see below, Values of Variables, section 3.2). To assign a value to a variable, you use the following notation:

```
dividend = 8;
divisor = 4;

myString = "I may want to use this message multiple times";
message = myString;
```

Let's say the main part of the function will be dividing the dividend by the divisor and storing that number in a variable called quotient. I can write this line of code in my program: quotient = divisor*dividend, and I have both stored the value of the quotient to the variable quotient and I have declared the variable at the same time. If I had wanted to, I could have declared it along with my other assigned variables above, with a value of null. After executing the program, the value of quotient will be 2.

It is important to think about the design of your program before you begin. You should create the appropriate variables so that it makes your life easier when you go to write the program. For instance, if you know that you will be coding a lot of the same strings in a message, you may want to create a variable called message and give it the value of your message. That way, when you call it in your program, you do not have to retype the same sentence over and over again, and if you want to change the content of that message, you only have to change it once -- in the variable declaration.

Values of Variables

JavaScript recognizes the following types of values:

- Numbers, such as 42 or 3.14159
- Logical (Boolean) values, either true or false
- Strings, such as "Howdy!"
- null, a special keyword which refers to nothing

This relatively small set of types of values, or data types, enables you to perform useful functions with your applications. There is no explicit distinction between integer and real-valued numbers.

Data Type Conversion

JavaScript is a loosely typed language. That means you do not have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42
```

And later, you could assign the same variable a string value, for example,

```
answer = "Thanks for all the fish..."
```

Because JavaScript is loosely typed, this assignment does not cause an error message. However, this is not good coding! You should create variables for a specific type, such as an integer, string, or array, and be consistent in the values that you store in the variable. This prevents confusion when you are writing your program.
In expressions involving numeric and string values, JavaScript converts the numeric values to strings. For example, consider the following statements:
x = "The answer is " + 42
y = 42 + " is the answer."
(The + sign tells JavaScript to concatenate, or stick together, the two strings. For example, if you write:
message = "Hello" + "World"
...then the variable message becomes the string "Hello World")
In the first statement, x becomes the string "The answer is 42." In the second, y becomes the string "42 is the answer."

Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you literally provide in your script. Examples of literals include: 1234, "This is a literal," and true.

Integers

Integers can be expressed in decimal (base 10), hexadecimal (base 16), and octal (base 8). A decimal integer literal consists of a sequence of digits without a leading 0 (zero). A leading 0 (zero) on an integer literal indicates it is in octal; a leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7.
Some examples of integer literals are: 42, 0xFFF, and -345.

Floating-point literals

A floating-point literal can have the following parts: a decimal integer, a decimal point ("."), a fraction (another decimal number), an exponent, and a type suffix. The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-"). A floating-point literal must have at least one digit, plus either a decimal point or "e" (or "E").
Some examples of floating-point literals are 3.1415, -3.1E12, .1e12, and 2E-12

Boolean literals

The Boolean type has two literal values: true and false.

String literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or double quotation marks. The following are examples of string literals:

- "blah"
- 'blah'
- "1234"
- "one line \n another line"

In addition to ordinary characters, you can also include special characters in strings, as shown in the last element in the preceding list. The following table lists the special characters that you can use in JavaScript strings.

Character	Meaning
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	tab

\\	backslash character

Escaping characters

For characters not listed in the preceding table, a preceding backslash is ignored, with the exception of a quotation mark and the backslash character itself.

You can insert quotation marks inside strings by preceding them with a backslash. This is known as escaping the quotation marks. For example,

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service."
document.write(quote)
```

The result of this would be

He read "The Cremation of Sam McGee" by R.W. Service.

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path c:\temp to a string, use the following:

```
var home = "c:\\temp"
```

Arrays

An Array is an object which stores multiple values and has various properties. When you declare an array, you must declare the name of it, and then how many values it will need to store. It is important to realize that each value is stored in one of the elements of the array, and these elements start at 0. This means that the first value in the array is really in the 0 element, and the second number is really in the first element. So for example, if I want to store 10 values in my array, the storage elements would range from 0-9.

The notation for declaring an array looks like this:

```
myArray = new Array(10); foo = new Array(5);
```

Initially, all values are set to null. The notation for assigning values to each unit within the array looks like this:

```
myArray[0] = 56;
myArray[1] = 23;
myArray[9] = 44;
```

By putting the element number in brackets [ ] after the array's name, you can assign a value to that specific element. Note that there is no such element, in this example, as myArray[10]. Remember, the elements begin at myArray[0] and go up to myArray[9].

In JavaScript, however, an array's length increases if you assign a value to an element higher than the current length of the array. The following code creates an array of length zero, then assigns a value to element 99. This changes the length of the array to 100.

```
colors = new Array();
colors[99] = "midnightblue";
```

Be careful to reference the right cells, and make sure to reference them properly!

Because arrays are objects, they have certain properties that are pre-defined for your convenience. For example, you can find out how many elements myArray has and store this value in a variable called numberOfElements by using:

```
numberOfElements = myArray.length;
```

Operators

JavaScript has many different operators, which come in several flavors, including binary. This tutorial will cover some of the most essential assignment, comparison, arithmetic and logical operators.

Selected assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The basic assignment operator is equal (=), which assigns the value of its right operand to its left operand. The other operators are shorthand for standard operations. Find an abridged list of shorthand operators below:

Shorthand operator	Meaning
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y

Note that the = sign here refers to assignment, not "equals" in the mathematical sense. So if x is 5 and y is 7, x = x + y is not a valid mathematical expression, but it works in JavaScript. It makes x the value of x + y (12 in this case).
EXAMPLES: If x = 10 and y = 2, x += y would mean x = x + y, hence x's new value would be the sum of x's previous value plus y's previous value. Upon executing x = x + y = 10 + 2, x's new value becomes 12, while y's new value remains equal to its old value. Similarly, x -= y would change x's value to 8. Calculate x *= y and x /= y to get a better idea of how these operators work.

Comparison operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true or not. The operands can be numerical or string values. When used on string values, the comparisons are based on the standard lexicographical ordering. They are described in the following table.

Operator	Description	Example
Equal (= =)	Evaluates to true if the operands are equal.	x == y evaluates to true if x equals y.
Not equal (!=)	Evaluates to true if the operands are not equal.	x != y evaluates to true if x is not equal to y.
Greater than (>)	Evaluates to true if left operand is greater than right operand.	x > y evaluates to true if x is greater than y.
Greater than or equal (>=)	Evaluates to true if left operand is greater than or equal to right operand.	x >= y evaluates to true if x is greater than or equal to y.
Less than (<)	Evaluates to true if left operand is less than right operand.	x < y evaluates to true if x is less than y.
Less than or equal (<=)	Evaluates to true if left operand is less than or equal to right operand.	x <= y evaluates to true if x is less than or equal to y.

EXAMPLES: 5 == 5 would return TRUE. 5 != 5 would return FALSE. (The statement 'Five is not equal to five.' is patently false.) 5 <= 5 would return TRUE. (Five is less than or equal to five. More precisely, it's exactly equal to five, but JavaScript could care less about boring details like that.)

Selected Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), division (/) and remainder (%). These operators work as they do in other programming languages, as well as in standard algebra.
Since programmers frequently need to add or subtract 1 from a variable, JavaScript has shortcuts for doing this. myVar++ adds one to the value of myVar, while myVar-- subtracts one from myVar.
EXAMPLES: Let x = 3. x++ bumps x up to 4, while x-- makes x equal to 2.

Logical Operators

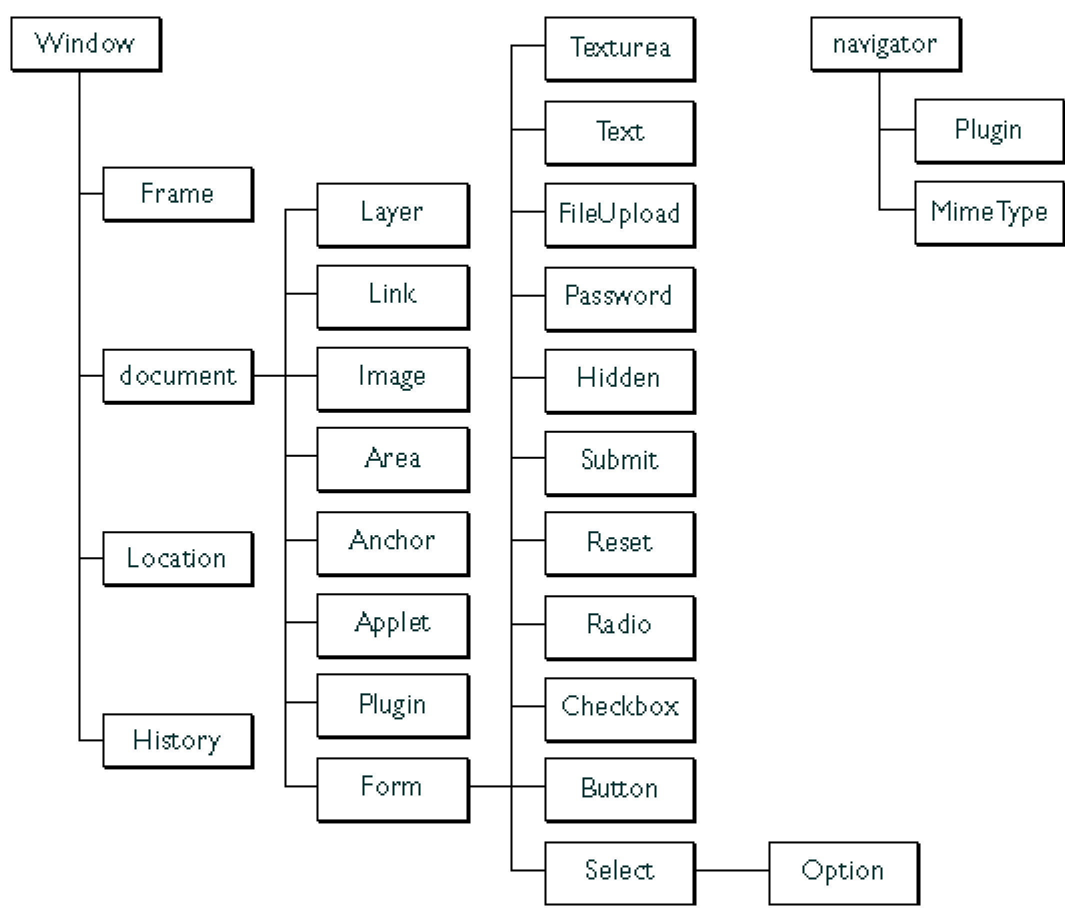
Logical operators take Boolean (logical) values as operands and return a Boolean value. That is, they evaluate whether each subexpression within a Boolean expression is true or false, and then execute the operation on the respective truth values. Consider the following table:

Operator	Usage	Description
and (&&)	expr1 && expr2	True if both logical expressions expr1 and expr2 are true. False otherwise.
or (||)	expr1 || expr2	True if either logical expression expr1 or expr2 is true. False if both expr1 and expr2 are false.
not (!)	!expr	False if expr is true; true if expr is false.

EXAMPLES: Since we have now learned to use the essential operators, we can use them in conjunction with one another. See if you can work out why the following examples resolve the way they do:
If x = 4 and y = 7, ((x + y + 2) == 13) && (((x + y) / 2) == 2) returns FALSE.
If x = 4 and y = 7, ((y - x + 9) == 12) || ((x * y) == 2) returns TRUE.
If x = 4 and y = 7, !((x/2 + y) == 9) || ((x * (y/2)) == 2) returns FALSE.

Using JavaScript Objects

When you load a document in your web browser, it creates a number of JavaScript objects with properties and capabilities based on the HTML in the document and other pertinent information. These objects exist in a hierarchy that reflects the structure of the HTML page itself.
The pre-defined objects that are most commonly used are the window and document objects. The window has methods that allow you to create new windows with the open() and close() methods. It also allows you to create message boxes using alert(), confirm(), and prompt(). Each displays the text that you put between the parentheses. For example, the following code:
alert("This is an alert box")
...pops up an alert box displaying the given message. Try it yourself by clicking on this link.
The document object models the HTML page. The document object contains arrays which store all the components constituting the contents of your web page, such as images, links, and forms. You can access and call methods on these elements of your web page through the arrays.
The objects in this pre-defined hierarchy can be accessed and modified. To refer to specific properties, you must specify the property name and all its ancestors, spelling out the complete hierarchy until the document object. A period, '.', is used in between each object and the name of its property. Generally, a property / object gets its name from the NAME attribute of the HTML tag. For example, the following refers to the *value* property of a text field named *text1* in a form named *myform* in the current document.
document.myform.text1.value
Form elements can also be accessed through the aforementioned forms array of the document object. In the above example, if the form named *myform* was the first form on the page, and *text1* was the third field in the form, the following also refers to that field's value property.
document.forms[0].elements[2].value
Functions (capabiltiies) of an object can similarly be accessed using the period notation. For example, the following instruction resets the 2nd form in the document.
document.forms[2].reset();
Click on one of the objects below to view the Netscape documentation on the specific properties and methods that that object has:



Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure -- a set of statements that performs a specific task. A function definition has these basic parts:

- The function keyword
- A function name
- A comma-separated list of arguments to the function in parentheses
- The statements in the function in curly braces: { }

Defining a Function

When defining a function, it is *very* important that you pay close attention to the syntax. Unlike HTML, JavaScript is case sensitive, and it is very important to remember to enclose a function within curly braces { }, separate parameters with commas, and use a semi-colon at the end of your line of code.

It's important to understand the difference between *defining* and *calling* a function.

Defining the function names the function and specifies what to do when the function is called. You define a function within the <SCRIPT>...</SCRIPT> tags within the <HEAD>...</HEAD> tags. In defining a function, you must also declare the variables which you will be calling in that function.

Here's an example of *defining* a function:

```
function popupalert() {

                alert('This is an alert box.');

}
```

Notice the parentheses after the function name. It is imperative that you include these parentheses, even if they are empty. If you want to pass a *parameter* into the function, you would include that parameter inside of the parentheses. A parameter is a bit of extra information that cn be different each time the function is run. It is stored in a variable and can be accessed just like any other variable. Here's an example of a function that takes a parameter:

```
function anotherAlert(word) {

        alert(word + ' is the word that you clicked on');

}
```

When you call this function, you need to pass a parameter (such as the word that the user clicked on) into the function. Then the function can use this information. You can pass in a different word as a parameter each time you call the function, and the alert box will change appropriately. You'll see how to pass a parameter a little later on.
You can pass in multiple parameters, by separating them with a comma. You would want to pass in a few parameters if you have more than one variable that you either want to change or use in your function. Here are two examples of passing in multiple parameters when you are defining the function:

```
function secondAlert(word,password) {

        confirm(word + ' is the word that you clicked on. The

                secret password is ' + password);

}
function thirdAlert(word,password) {

        confirm(word + ' is the word you clicked on. Please

                take note of the password, ' + password);

}
```

You'll notice that the same parameters are passed into both of these functions. However, you can pass in whatever values you want to use (see this same example below in calling the function).

Calling a Function

Calling the function actually performs the specified actions. When you call a function, this is usually within the BODY of the HTML page, and you usually pass a parameter into the function. A parameter is a variable from outside of the defined function on which the function will act.
Here's an example of calling the same function:
popupalert();
For the other example, this is how you may call it:

```
<A HREF="#top" onClick="anotherAlert('top')">top</A>
```

This would bring you to the top of the page, and bring up an alert box that said: "top is the word you clicked on" Try it for yourself: top

Here is the same example with multiple parameters that was shown above:

```
<A HREF="#top" onClick="secondAlert('awesome','pandas')">awesome</A>

<A HREF="#top" onClick="thirdAlert('computers','insert')">computers</A>
```

You'll notice in the code that different values for the variables word and password are passed in. These values here are what the function will need to perform the actions in the function. Make sure that the values you pass in are in the correct order because the function will take them in and assign these values to the parameters in the parentheses of the function declaration. Once you pass values into your function, you can use them however you want within your function.

Try it for yourself:

When you click on the words below, a confirmation box will pop up and then the link will bring you to the top of the page.

awesome

computers

If/Else Statements

if statements execute a set of commands if a specified condition is true. If the condition is false, another set of statements can be executed through the use of the else keyword.

The main idea behind if statements is embodied by the sentence: "If the weather's good tomorrow, we'll go out and have a picnic and Lisa will do cartwheels -- else, we'll stay in and Catherine will watch TV."

As you can see, the idea is quite intuitive and, surprisingly enough, so is the syntax:

```
if (condition) {
        statements1
}
```

-or-

```
if (condition) {
        statements1
}
else {
        statements2
}
```

(An **if** statement does not require an **else** statement following it, but an **else** statement must be preceded by an **if** statement.)

condition can be any JavaScript expression that evaluates to true or false. Parentheses are required around the condition. If *condition* evaluates to true, the statements in *statements1* are executed.

statements1 and *statements2* can be any JavaScript statements, including further nested **if** statements. Multiple statements must be enclosed in braces.

Here's an example:

```
if (weather == 'good') {
        go_out(we);
        have_a_picnic(we);
        do_cartwheels(Lisa);
}
else {
        stay_in(we);
```

```
        watch_TV(Catherine);
}
```

Loops

Loops are an incredibly useful programming tool. Loops handle repetitive tasks extremely well, especially in the context of consecutive elements. Arrays immediately spring too mind here, since array elements are numbered consecutively. It would be quite intuitive (and equally practical), for instance, to write a loop that added 1 to each element within an array. Don't worry if this doesn't make a lot of sense now, it will, after you finish reading the tutorial.

The two most common types of loops are for and while loops:

for Loops

A for loop constitutes a statement which consists of three expressions, enclosed in parentheses and separated by semicolons, followed by a block of statements executed in the loop.

This definition may, at first, sound confusing. Indeed, it is hard to understand for loops without seeing them in action.

A for loop resembles the following:

```
for (initial-expression; condition; increment-expression) {
        statements
}
```

The *initial-expression* is a statement or variable declaration. (See the section on variables for more information.) It is typically used to initialize a counter variable. This expression may optionally declare new variables with the var keyword.

The *condition* is evaluated on each pass through the loop. If this condition evaluates to true, the statements in *statements* are performed. When the condition evaluates to false, the execution of the for loop stops. This conditional test is optional. If omitted, the condition always evaluates to true.

The *increment-expression* is generally used to update or increment the counter variable.

The *statements* constitute a block of statements that are executed as long as *condition* evaluates to true. This can be a single statement or multiple statements. Although not required, it is good practice to indent these statements from the beginning of the for statement to make your code more readable.

Check out the following for statement. It starts by declaring the variable *i* and initializing it to zero. It checks whether *i* is less than nine, performs the two successive statements, and increments *i* by one after each pass through the loop:

```
var n = 0;
for (var i = 0; i < 3; i++) {
        n += i;
        alert("The value of n is now " + n);
}
```

while Loops

The while loop, although most people would not recognize it as such, is for's twin. The two can fill in for one another - using either one is only a matter of convenience or preference according to context. while creates a loop that evaluates an expression, and if it is true, executes a block of statements. The loop then repeats, as long as the specified condition is true.

The syntax of while differs slightly from that of for:

```
while (condition) {
        statements
}
```

condition is evaluated before each pass through the loop. If this condition evaluates to true, the statements in the succeeding block are performed. When *condition* evaluates to false, execution continues with the statement following *statements*.

statements is a block of statements that are executed as long as the *condition* evaluates to true. Although not required, it is good practice to indent these statements from the beginning of the statement.
The following while loop iterates as long as *n* is less than three.

```
var n = 0;
var x = 0;
while(n < 3) {
        n++;
        x += n;
        alert("The value of n is " + n + ". The value of x is " + x);
}
```

Try it for yourself: Click this link

Commenting

Comments allow you to write notes to yourself within your program. These are important because they allow someone to browse your code and understand what the various functions do or what your variables represent. Comments also allow you to understand your code if it's been a while since you last looked at it.

In JavaScript, you can write both one-line comments and multiple line comments. The notation for each is different though. For a one line comment, you precede your comment with //. This indicates that everything written on that line, after the //, is a comment and the program should disregard it.
For a multiple-line comment, you start with /* and end with */ . It is nice to put an * at the beginning of each line just so someone perusing your code realizes that he/she is looking at a comment (if it is really long this helps). This is not necessary though.
The following are examples of comments in JavaScript.

```
// This is a single line comment.

/* This is a multiple line comment with only one line. */

/* This is a multiple line comment.
 * The star (*) at the beginning of this line is optional.
 * So is the star at the beginning of this line. */
```

JavaScript program using objects

```
<html>
<head>
<script language=”javascript”>
function demo1()
{
Popup(“Hello”);
Obj= new sample (2, 4);
alert(obj.x + obj.y);
}
function sample(x,y)
{
this.x=x;
this.y=y;
}
</script>
</head.
```

```
<body onLoad="demo1( )">
</body>
</html>
```

Regular Expression

A script language may take name data from a user and have to search through the string one character at a time. The usual approach in scripting language is to create a pattern called a regular expression which describes a set of characters that may be present in a string.

```
    var pattern = "target";
  var string = "can you find the target";
    string.match(pattern);
```

But the above code can also be written using regular expression as a parameter, as shown below.

```
    var pattern = new RegExp("target");
    var string = "can you find the target";
    pattern.exec(string);
```

Regular expression is a javascript object. Dynamic patterns are created using the keyword new.

```
    regex = new RegExp("feroz | btech");
```

JavaScript code to implement RegExp

```
<html>
<head>
<body>
<script language="javascript">
var re = new RegExp("[A | a]mer");
var msg=" Have you met Btech recently";
var res= re.exec(msg);
if(res)
{
alert( " I found " + res[0]);
}
else
{
alert(" I didn't find it");
}
</script>
</body>
</html>
```

Functions:

Regular Expressions are manipulated using the functions which belong to either the RegExp or String class.

Class String functions

match(pattern)

This function searches a matching pattern. Returns array holding the results.

replace(pattern1, pattern2)

Searches for pattern1. If the search is successful pattern1 is replaced with pattern2.

search(pattern)

Searches for a pattern in the string. If the match is successful, the index of the start of the match is returned. If the search fails, the function returns -1.

Class RegExp functions
exec(string)
Executes a search for a matching pattern in its parameter string. Returns an array holding the results of the operation.
test(string)
Searches for a match in its parameter string. Returns true if a match is found, otherwise returns false.

Built in objects:

The document object

A document is a web page that is being either displayed or created. The document has a number of properties that can be accessed by JavaScript programs and used to manipulate the content of the page.
Write or writeln
Html pages can be created using JavaScript. This is done by using the write or writeln methods of the document object.

```
Document.write("<body>");
Document.write("<h1> Hello </h1>");
```

The form object
Two aspects of the form can be manipulated through JavaScript. First, most commonly and probably most usefully, the data that is entered onto your form can be checked at submission. Second you can actually build forms through JavaScript.

Example : Validate.js

```
function validate()
{
var t1=document.forms[0].elements;
var t2=parent.frames['f4'].document;
var bg1=t1.bg.value;
var c1=t1.c.value;
t2.open();
t2.write("<body bgcolor="+bg1+">");
t2.write("Candidate name is : "+c1);
t2.write("</body>");
t2.close();
}
```

Mypage.html

```
<html>
<head>
<script language = "javascript src= "D:\Documents and Settings \ p6 \ validate.js">
</script>
</head>
<body>
<form>
Background Color: <input type="text" size=16 name="bg" value="white">
Candidate's name:<input type="text" size=16 name="c">
```

```
<input type="button" value="showit" onClick="validate()">
</form>
</body>
</html>
```

The browser object

Some of the properties of the browser object is as follows

Navigator.appCodeName : The internal name for the browser.
Navigator.appVersion:This is the public name of the browser.
Navigator.appVersion:The version number, platform on which the browser is running.
Navigator.userAgent :The strings appCodeName and appVersion concatenated together.

The Date object
JavaScript provides functions to perform many different date manipulation. Some of the functions are mentioned below.
Date( ) : Construct an empty date object.
Date(year, month, day [,hour, minute, second]) :Create a new Date object based upon numerical values for the year, month and day. Optional time values may also be supplied.
getDate( ):Return the day of the month
getDay( ):Return an integer representing the day of the week.
getFullYear( ):Return the year as a four digit number.
getHours( ):Return the hour field of the Date object.
getMinutes( ):Return the minutes field of the Date object.
getSeconds( ):Return the second field of the Date object.
setDate(day):Set the day value of the object. Accepts values in the range 1 to 31.
setFullYear(year [,month, day]):Set the year value of the object. Optionally also sets month and day values.
toString( ):Returns the Date as a string.

Events:
JavaScript is a event-driven system. Nothing happens unless it is initiated by an event outside the script. The table below shows event, event handler and the description about the event handler.
The following are events used with the elements

Event Attribute	Description	Tags/elements used
onblur	When the field lost focuses and Enters into another field usually By tag or mouse click	Area,button,input,select, textarea
onchange	Whenever the text or options are Modified	Input,select,textarea
onclick	On clicking the button or reset or submit etc	Most elements
ondblclick	Clicking twice	Most elements
onfocus	Got focus in the field or entering into the field	Area,button,input,select, textarea
Onkeydown	Key pressed down	Most elements
onkeyup	Pressing up	Most elements

onload	When the document is loaded	Body,frameset
Onmousedown	Moving mousedown	Most elements
Onmousemove	Moving mouse	Most element
Onmouseout	Mouse moved away	Mostelement
Onmouseover	Placingmouse on it	Most
Onreset	Clicking on reset button	Form
Onselect	Selecting text	Input.textarea
Onsubmit	Clicking on submit button	Form
Onunload	When unloaded from memory	Body ,frameset

Dynamic HTML with JavaScript

Data Validation

Data validation is the common process that takes place in the web sites. One common request is for a way of validating the username and password. Following program shows the validation of data which uses two frames, in one frame user is going to enter the data and in the other frame equivalent result is going to be displayed.

Example JavaScript code for data validation

Mypage.html

```
<html>
<head>
<title>frame page </title>
</head>
<frameset rows="20%,*">
<frame name="f1" src="">
<frameset cols="20%,*">
<frame name="f2" src="">
<frameset cols="50%,*">
<frame name="f3" src="D:\Documents and Settings\Btech\Desktop\btech\p6\reg.html">
<frame name="f4" src="D:\Documents and
Settings\Btech\Desktop\btech\p6\profile.html">
</frameset>
</frameset>
</frameset>
</html>
```

Myform.html

```
<html>
<head>
<script language = "javascript" src = "D:\ Documents and Settings \ Btech\ Desktop\btech\ p6\ validate.js">
</script>
</head>
<body>
<form>
Background Color: <input type="text" size=16 name="bg" value="white">
```

```
Candidate's name:<input type="text" size=16 name="c">
<input type="button" value="showit" onClick="validate()">
</form>
</body>
</html>
```

Validate.js

```
function validate()
{
var t1=document.forms[0].elements;
var t2=parent.frames['f4'].document;
var bg1=t1.bg.value;
var c1=t1.c.value;
t2.open();
t2.write("<body bgcolor="+bg1+">");
t2.write("Candidate name is : "+c1);
t2.write("</body>");
t2.close();
}
```

UNIT-III: XML(extensible markup language)

CONTENTS

- Document type definition,
- XML Schemas,
- Document Object model
- Presenting XML
- Using XML Processors: DOM and SAX

UNIT III

XML

The markup language developed to add structural and formatting information to data and which was designed to be simple enough to be included in any application that language is Standard Generalized Markup Language and was adopted as standard by International Organization for Standardization(ISO).
Markup is nothing but instructions, which are often called as tags. There are many languages which shows how the data is displayed but no one describes what the data is.This is the point at which XML enters. XML is a subset of SGML. XML is used to describe the structure of a document not the way that is presented. XML is the recommendation of World Wide Consortium (W3C). The structure of basic XML is
shown below which resembles HTML. The first line is the processing instruction which tells applications how to handle the XML. It is also serves as version declaration and says that the file is XML.

Example Sample XML program

```
<?xml version="1.0"?>
<college>
<studdetail>
        <regno>05j0a1260</regno>
        <name>
        <firstname>karthik></firstname>
        <lastname>btech</lastname>
        </name>
        <country name="india"/>
        <branch>csit</branch>
</studdetail>
</college>
```

Valid an Well Formed XML

XML documents may be either valid or well formed. A well formed document is one which follows all of the rules of XML. Tags are matched and do not overlap, empty elements are ended properly, and the document contains an XML declaration. A valid XML document has its own DTD. XML should also conforms the rules set out in the DTD. There are many XML parsers that checks the document and its DTD

XML elements

XML documents are composed of three things i.e., elements, control information, and entities. Most of the markup in an XML document is element markup. Elements are surrounded by tags much as they are in HTML. Each document has a single root element which contains al of the other markup.

Nesting tags: Even the simplest XML document has nested tags. Unlike HTML these must be properly nested and closed in the reverse of the order in which they were opened. Each XML tag has to have a closing tag, again unlike HTML.

Case Sensitive: XML is case sensitive and you must use lower case for your markup.

Empty tags: Some tags are empty, they don't have content. Where the content of the tag is missing, it appears as <content/>

Attributes: Sometimes it is important that elements have information associated with them without that information becoming a separate element.

Control Information:

There are three types of control information

- Comments
- processing instructions

- document type declaration.

Comments: XML comments are exactly same as HTML. They take the form as
<!- -comment text - ->
Processing Instructions: Processing Instructions are used to control applications. One of the processing instructions is **<?xml version=”1.0”>**
Document Type Declarations: Each XML document has an associated DTD. The DTD is usually present in separate file, so that it can be used by many files. The statement that includes DTD in XML file is **<?DOCTYPE cust SYSTEM “customer.dtd”>**

Entities

Entities are used to create small pieces of data which you want to use repeatedly throughout your schema.

Example A Complete XML program

```
<?xml version=”1.0”?>
<!DOCTYPE stud S?YSTEM “student.dtd”>
<college>
<studdetail>
<regno>05j0a1260</regno>
<name>
<firstname>feroz></firstname>
<lastname>btech</lastname>
</name>
<country name="india"/>
<branch>csit</branch>
</studdetail>
</college>
```

Document Type Definition

Document type definition have been successfully used in SGML applications for many year. DTD are document centric. They are well understood. There are plenty of tools that support DTD.

```
<!ELEMENT college(studetail+)>
<!ELEMENT studetail(regno, name+, country, branch)>
<!ELEMENT regno(#PCDATA)>
<!ELEMENT name(firstname, lastname)>
<!ELEMENT firstname(#PCDATA)>
<!ELEMENT lastname(#PCDATA)>
<!ELEMENT country(#PCDATA)>
<!ATTLIST country name CDATA #REQUIRED>
<!ELEMENT branch(#PCDATA)>
```

XML Schema

W3C developed a technology called XML schema which they accepted as a recommendation. XML schema is itself an XML application which means when you use it your only need a single grammar and can use your normal XML editor to create it.

Example XML Schema for XML document shown in Example

```
<?xml version =”1.0” ?>
<xsd:schema xmlns =” http://.........”>
<xsd:element name = “college”>
<xsd:complexType>
<xsd:sequence>
```

```
<xsd:element name = “studetail”>
<xsd:complexType>
<xsd:sequence>
<xsd:element name = “regno” type = “xsd:string”/>
<xsd:element name = “name”>
<xsd:complexType>
<xsd:sequence>
<xsd:element name = “firstname” type=”xsd:string”/>
<xsd:element name= “lastname” type = “xsd:string”/>
<xsd:element name = “country”>
<xsd:complexType>
<xsd:attribute name = “India” type= “xsd:string”/>
</xsd:complexType>
</xsd:element>
xsd:element name = “branch” type = “xsd:string”/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Document Object Model

XML parsers can handle documents in any way that their developers choose. There are two models commonly used for parsers i.e., SAX and DOM. SAX parsers are used when dealing with streams of data. This type of parsers are usually used with java.

- SAX-based parsers run quickly.
- DOM is and application program interface (API) for XML documents.

The DOM API specifies the logical structure of XML documents and the ways in which they can be accessed and manipulated. The DOM API is just a specification. DOM-complaint applications include all of the functionality needed to handle XML documents. They can build static documents, navigate and search through them, add new elements, delete elements, and modify the content of existing elements. The views XML document as trees. The DOM exposes the whole of the document to applications. It is also scriptable so applications can manipulate the individual nodes.

Presenting XML

XML documents are presented using Extensible Stylesheet which expresses stylesheets.
XSL stylesheet are not the same as HTML cascading stylesheets. They create a style for a specific XML element, with XSL a template is created. XSL basically transforms one data structure to another i.e., XML to HTML.

Example Here is the XSL file for the XML document of Example

This line must be included in the XML document which reference stylesheet

```
<?xml:stylesheet type = “text/xsl” href = “student.xsl”?.
```

Here goes the XSL file

```
<xsl:stylesheet smlns:xsl =”uri:xsl”.
<xsl:template match=”/”>
<html>
<body>
<h1> Student Database </h1.
<xsl:for-each select = “college”>
<xsl:for-each select = “studetail”>
<xsl:value-of select = “regno”/>
<xsl:for-each select = “name”>
<xsl:value-of select = “firstname”/>
<xsl:value-of select = “lastname”/>
</xsl:for-each>
<xsl:value-of select=”country/@name” />
<xsl:value-of select = “branch”/>
</xsl:for-each>
</xsl:for-each>
</body>
</xsl:template>
</xsl:stylesheet>
```

Evolution of the XML Parsing

The combination of Java and XML has been one of the most attracting things which had happened in the field of software development in the 21st century. It has been mainly for two reasons - Java, arguably the most widely used programming language and XML, almost unarguably the best mechanism of data description and transfer.

Since these two were different technologies and hence it initially required a developer to have a sound understanding of both of these before he can make the best use of the combination. Since then there have been a paradigm shift towards Java a few interesting technologies getting evolved to make this happen. Some of them are:-

SAX - Simple API for XML Parsing

It was the first to come on the scene and interestingly it was developed in the XML-Dev maling list. Evidently the people who developed this were XML gurus and it is quite visible in the usage of this API. You got to have a fair understanding of XML, but at least Java developers got something to combine the two worlds - Java and XML in a structured way. It instantly became a hit for the obvious reasons.

Since this API does require to load the entire XML doc and also because it offers only a sequential processing of the doc hence it is quite fast. Another reason of it being faster is that it does not allow modification of the underlying XML data.

DOM - Document Object Model

The Java binding for DOM provided a tree-based representation of the XML documents - allowing random access and modification of the underlying XML data. Not very difficult to deduce that it would be slower as compared to SAX.

The event-based callback methodology was replaced by an object-oriented in-memory representation of the XML documents. Though, it differs from one implementation to another if the entire document or a part of it would be

kept in the memory at a particular instant, but the Java developers are kept out of all the hassle and they get the entire tree readily available whenever they wish.

JAXP - Java API for XML Parsing

The creators and designers of Java realized that the Java developers should not be XML gurus to use the XML in Java applications. The first step towards making this possible was the evolution of JAXP, which made it easier to obtain either a DOM Document or a SAX-compliant parser via a factory class. This reduced the dependence of Java developers over the numerous vendors supplying the parsers of either type. Additionally, JAXP made sure that an interchange between the parsers required minimal code changes.

Differences between DOM and SAX

SAX v/s DOM

Main differences between SAX and DOM, which are the two most popular APIs for processing XML documents in Java, are:-

- **Read v/s Read/Write:** SAX can be used only for reading XML documents and not for the manipulation of the underlying XML data whereas DOM can be used for both read and write of the data in an XML document.
- **Sequential Access v/s Random Access:** SAX can be used only for a sequential processing of an XML document whereas DOM can be used for a random processing of XML docs. So what to do if you want a random access to the underlying XML data while using SAX? You got to store and manage that information so that you can retrieve it when you need.
- **Call back v/s Tree:** SAX uses call back mechanism and uses event-streams to read chunks of XML data into the memory in a sequential manner whereas DOM uses a tree representation of the underlying XML document and facilitates random access/manipulation of the underlying XML data.
- **XML-Dev mailing list v/s W3C:** SAX was developed by the XML-Dev mailing list whereas DOM was developed by W3C (World Wide Web Consortium).
- **Information Set:** SAX doesn't retain all the info of the underlying XML document such as comments whereas DOM retains almost all the info. New versions of SAX are trying to extend their coverage of information.

UNIT –IV : JAVA BEANS

CONTENTS

- Introduction to Java Beans
- Advantages of Java Beans
- BDK Introspection
- Using Bound properties
- Bean Info Interface
- Constrained properties
- Persistence, Customizes
- Java Beans API
- Introduction to EJB’s

UNIT IV

Introduction to Java Beans

A Java Beans is software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform simple function, such as checking the spelling of a document, or complex function, such as forecasting the performance of a stock portfolio. A bean may be visible to an end user. One example of this is a button on a graphical user interface. A bean may be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components.

Advantages of Java Beans

- A bean obtains all the benefits of Java's "write once, run-anywhere" paradigm.
- The properties, events and methods of a bean that are exposed to an application builder tool can be controlled.
- A bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a bean.
- The configuration settings of a bean can be saved in persistent storage and restored at a later time.
- A bean may register to receive events from other objects and can generate events that are sent to other objects.

BDK Introspection

Introspection is the process of analyzing a bean to determine its capabilities. This is a very important feature of Java Bean API, because it allows an application builder tool to present information about a component to a software designer. Without introspection, the java beans technology could not operate. One way exposed the properties, events and methods of bean to application builder tool is using simple naming conventions.

Design pattern for properties

Property is a subset of a bean's state. The values that are assigned to the properties determine the behavior and appearance of that component.

Simple properties:

A simple property has a single value. It can be identified by the following design patterns, where N is the name of the property and T is its type.

```
Public T getN( );
Public void setN( );
```

Boolean properties:

A Boolean property has a value of true or false. It can be identified by the following design patterns, where N is name of the property.

```
Public Boolean isN ( );
Public Boolean getN( );
Public void setN(Boolean value);
```

indexed properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where N is the name of the property and T is its type.

```
Public T getN(int index);
Public void setN(int index, T value);
Public T[ ] getN( );
Public void setN(T values[ ]);
```

Using Bound Properties

A bean that has a bound property generates an event when the property is changed. The event is of type PropertyChangeEvent and is sent to objects that previously registered an interest in receiving such notifications.

Example : Application that uses TickTock bean to automatically control the Color bean

Steps:

1. Go to menu bar of the bean box and select Edit | Events | propetyChange. We can see a line extending from the button to the cursor
2. Move the cursor so that it is inside the Colors bean display area , and click the left mouse button. See the Event Target Dialog dialog box. dialog box allows to choose a method that should be invoked when this event occurs. Select the entry labeled "change" and click the Ok button.

Using BeanInfo Interface

This interface defines several methods, including these:

PropertyDescription[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )

The above methods will return array of objects that provide information about the properties, events, and methods of bean. SimpleBeanInfo is a class that provides default implementations of the BeanInfo interface, including the three methods . this class and override on or more of them.

Constrained Properties

A bean that has a constrained property generates an event when an attempt is made to change its value. The event is of type PropertyChangeEvent. It is sent to objects that previously registered an interest in receiving such notifications. This capability allows a Bean to operate differently according to its run-time environment.

Persistence

Persistence is the ability to save a Bean to nonvolatile storage and retrieve it at a later time. The information that is particularly important are the configuration settings.

Customizers

A bean developer can provide a customizer that helps another developer configure this software. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context.

Java Beans API

Interface Description

AppletInitializer Methods present in this interface are used to initialize Beans that are also applets

BeanInfo This interface allows a designer to specify information about the properties, events and methods of a Bean.

Customizer This interface allows a designer to provide a graphical user interface through which a Bean may be configured.

DesignMode Methods in this interface determine if a Bean is executing in design mode.

PropertyChangeListener A method in this interface is invoked when a bound property is changed.

Visibility Methods in this interface allow a bean to execute in environments where graphical user interface is not available.

Class Description

BeanDescriptor This class provides information about a Bean.

Beans This class is used to obtain information about a Bean

IntrospectionException An exception of this type is generated if a problem occurs when analyzing a bean.

PropertyChangeEvent This event is generated when bound or constrained properties are changed.

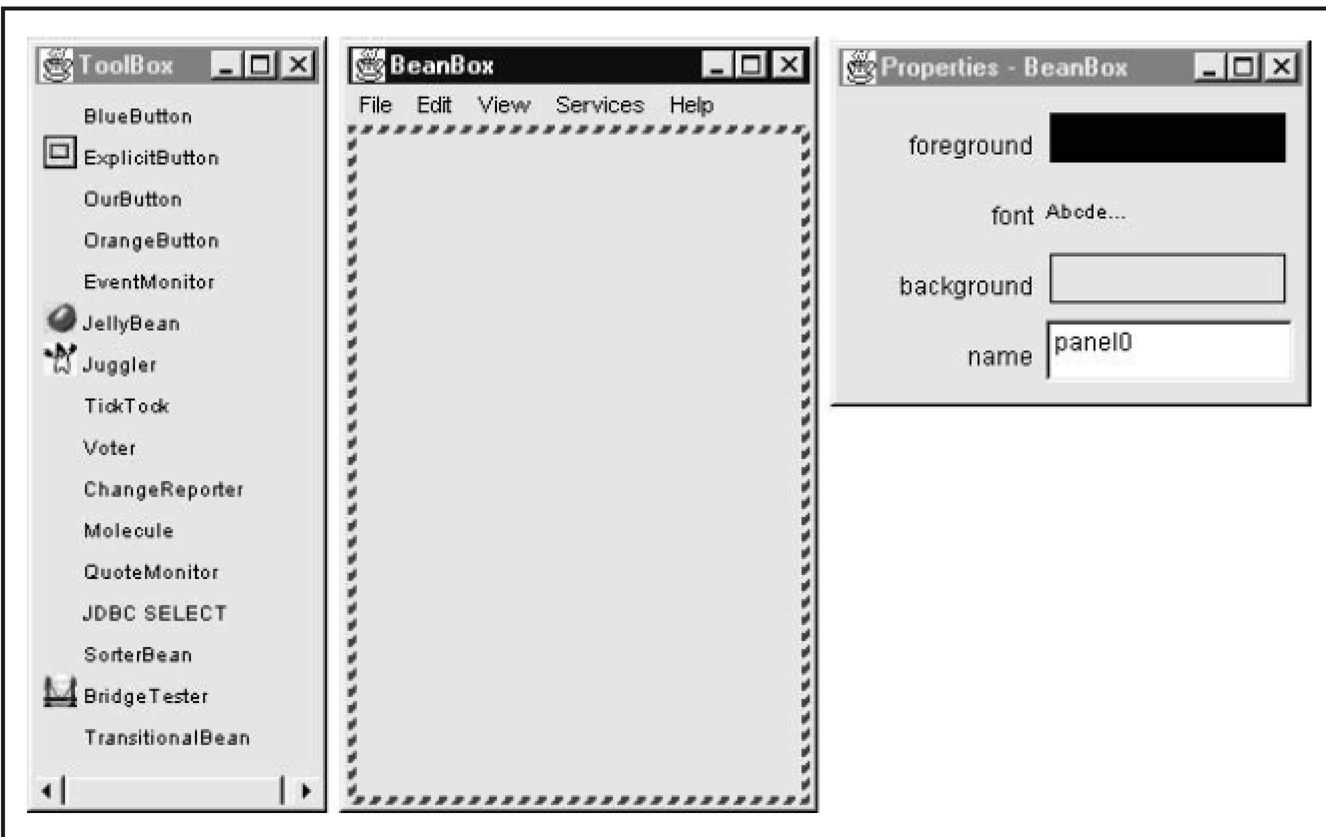
PropertyDescriptor Instances of this class describe a property of a Bean

Sun provides two Bean application builder tools. The first is the BeanBox, which is part of the Bean Developers Kit (BDK). The BDK is the original builder tool provided by Sun. The second is the new Bean Builder. Because Bean Builder is designed to supplant the BeanBox, Sun has stopped development of the BDK and all new Bean applications will be created using Bean Builder.

Using the Bean Developer Kit (BDK)

The Bean Developer Kit (BDK), available from the JavaSoft site, is a simple example of a tool that enables you to create, configure, and connect a set of Beans. There is also a set of sample Beans with their source code. This section provides step-by-step instructions for installing and using this tool.

Starting the BDK



To start the BDK, follow these steps:

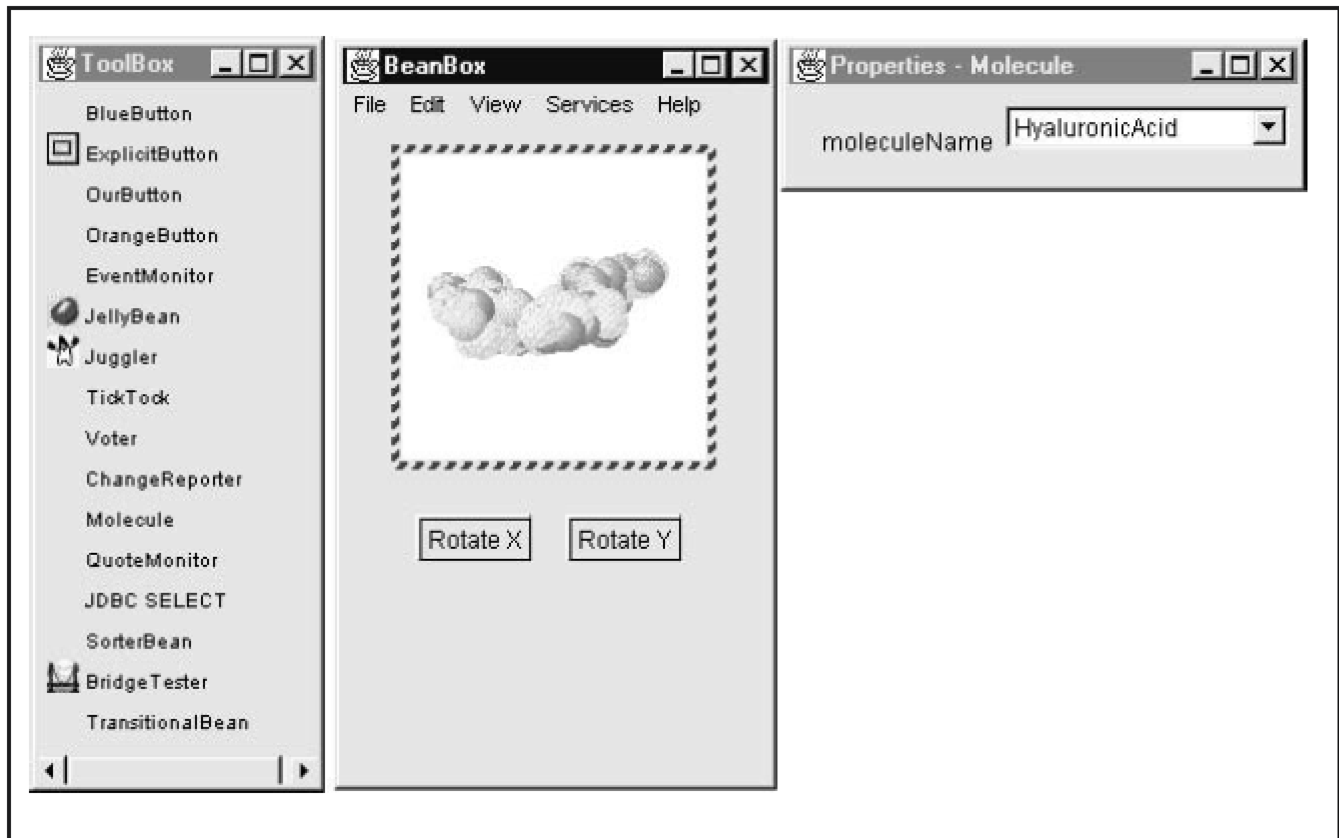
1. Change to the directory **c:\bdk\beanbox**.
2. Execute the batch file called **run.bat**. This causes the BDK to display the three windows shown in Figure . ToolBox lists all of the different Beans that have been included with the BDK. BeanBox provides an area to lay out and connect the Beans selected from the ToolBox. Properties provides the ability to configure a selected Bean. You may also see a window called Method Tracer,

Using the BDK

First, the **Molecule** Bean displays a three-dimensional view of a molecule. It may be configured to present one of the following molecules: hyaluronic acid, benzene, buckminsterfullerine, cyclohexane, ethane, or water. This component also has methods that allow the molecule to be rotated in space along its X or Y axis.

Second, the **OurButton** Bean provides a push-button functionality. We will have one button labeled “Rotate X” to rotate the molecule along its X axis and another button labeled “Rotate Y” to rotate the molecule along its Y axis.

Create and Configure an Instance of the Molecule Bean



Follow these steps to create and configure an instance of the **Molecule** Bean:

1. Position the cursor on the ToolBox entry labeled **Molecule** and click the left mouse button. You should see the cursor change to a cross.
2. Move the cursor to the BeanBox display area and click the left mouse button in approximately the area where you wish the Bean to be displayed. You should see a rectangular region appear that contains a 3-D display of a molecule. This area is surrounded by a hatched border, indicating that it is currently selected.
3. You can reposition the **Molecule** Bean by positioning the cursor over one of the hatched borders and dragging the Bean.
4. You can change the molecule that is displayed by changing the selection in the Properties window. Notice that the Bean display changes immediately when you change the selected molecule.

Create and Configure an Instance of the OurButton Bean
Follow these steps to create and configure an instance of the **OurButton** Bean and connect it to the **Molecule** Bean:
1. Position the cursor on the ToolBox entry labeled **OurButton** and click the left mouse button. You should see the cursor change to a cross.
2. Move the cursor to the BeanBox display area and click the left mouse button in approximately the area where you wish the Bean to be displayed. You should see a rectangular region appear that contains a button. This area is surrounded by a hatched border indicating that it is currently selected.
3. You may reposition the **OurButton** Bean by positioning the cursor over one of the hatched borders and dragging the Bean.
4. Go to the Properties window and change the label of the Bean to "Rotate X". The button appearance changes immediately when this property is changed.
5. Go to the menu bar of the BeanBox and select Edit | Events | action |actionPerformed. You should now see a line extending from the button to the cursor. Notice that one end of the line moves as the cursor moves. However, the other end of the line remains fixed at the button.
6. Move the cursor so that it is inside the **Molecule** Bean display area, and click the left mouse button. You should see the Event Target Dialog dialog box.
7. The dialog box allows you to choose a method that should be invoked when this button is clicked. Select the entry labeled "rotateOnX" and click the OK button. You should see a message box appear very briefly, stating that the tool is "Generating and compiling adaptor class."

Test the application. Each time you press the button, the molecule should move a few degrees around one of its axes.
Now create another instance of the **OurButton** Bean. Label it "Rotate Y" and map its action event to the "rotateY" method of the **Molecule** Bean. The steps to do this are very similar to those just described for the button labeled "Rotate X".
Test the application by clicking these buttons and observing how the molecule moves.

JAR Files

Before developing your own Bean, it is necessary for you to understand JAR (Java Archive) files, because tools such as the BDK expect Beans to be packaged within JAR files. A JAR file allows you to efficiently deploy a set of classes and their associated resources. For example, a developer may build a multimedia application that uses various sound and image files. A set of Beans can control how and when this information is presented. All of these pieces can be placed into one JAR file.
JAR technology makes it much easier to deliver and install software. Also, the elements in a JAR file are compressed, which makes downloading a JAR file much faster than separately downloading several uncompressed files. Digital signatures may also be associated with the individual elements in a JAR file. This allows a consumer to be sure that these elements were produced by a specific organization or individual.
The package java.util.zip contains classes that read and write JAR files.

Manifest Files

A developer must provide a *manifest file* to indicate which of the components in a JAR file are Java Beans. An example of a manifest file is provided in the following listing. It defines a JAR file that contains four **.gif** files and one **.class** file. The last entry is a Bean.

```
Name: sunw/demo/slides/slide0.gif
Name: sunw/demo/slides/slide1.gif
Name: sunw/demo/slides/slide2.gif
Name: sunw/demo/slides/slide3.gif
Name: sunw/demo/slides/Slides.class
Java-Bean: True
```

A manifest file may reference several **.class** files. If a **.class** file is a Java Bean, its entry must be immediately followed by the line "Java-Bean: True".

The JAR Utility
A utility is used to generate a JAR file. Its syntax is shown here:
jar *options files*

Option	Description
c	A new archive is to be created.
C	Change directories during command execution.
f	The first element in the file list is the name of the archive that is to be created or accessed.
i	Index information should be provided.
m	The second element in the file list is the name of the external manifest file.
M	Manifest file not created.
t	The archive contents should be tabulated.
u	Update existing JAR file.
v	Verbose output should be provided by the utility as it executes.
x	Files are to be extracted from the archive. (If there is only one file, that is the name of the archive, and all files in it are extracted. Otherwise, the first element in the file list is the name of the archive, and the remaining elements in the list are the files that should be extracted from the archive.)
0	Do not use compression.

Creating a JAR File
The following command creates a JAR file named **Xyz.jar** that contains all of the **.class** and **.gif** files in the current directory:

jar cf Xyz.jar *.class *.gif

If a manifest file such as **Yxz.mf** is available, it can be used with the following command:

jar cfm Xyz.jar Yxz.mf *.class *.gif

The following command lists the contents of **Xyz.jar**:
jar tf Xyz.jar

Introspection

Introspection is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API, because it allows an application builder tool to present information about a component to a software designer. Without introspection, the Java Beans technology could not operate. There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed by an application builder tool. With the first method, simple naming conventions are used. These allow the introspection

mechanisms to infer information about a Bean. In the second way, an additional class is provided that explicitly supplies this information.

Design Patterns for Properties

A *property* is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. This section discusses three types of properties: simple, Boolean, and indexed.

Simple Properties

A simple property has a single value. It can be identified by the following design patterns, where N is the name of the property and T is its type.

```
public T getN( );
public void setN(T arg);
```

A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

The following listing shows a class that has three read/write simple properties:

```
public class Box {
private double depth, height, width;
public double getDepth( ) {
return depth;
}
public void setDepth(double d) {
depth = d;
}
public double getHeight( ) {
return height;
}
public void setHeight(double h) {
height = h;
}
public double getWidth( ) {
return width;
}
public void setWidth(double w) {
width = w;
}
}
```

Boolean Properties

A Boolean property has a value of **true** or **false**. It can be identified by the following design patterns, where N is the name of the property:

```
public boolean isN( );
public boolean getN( );
public void setN(boolean value);
```

Either the first or second pattern can be used to retrieve the value of a Boolean property. However, if a class has both of these methods, the first pattern is used.

Indexed Properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where N is the name of the property and T is its type:

```
public T getN(int index);
public void setN(int index, T value);
public T[ ] getN( );
public void setN(T values[ ]);
```

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans can generate events and send them to other objects. These can be identified by the following design patterns, where T is the type of the event:

```
public void addTListener(TListener eventListener);
public void addTListener(TListener eventListener) throws TooManyListeners;
public void removeTListener(TListener eventListener);
```

These methods are used by event listeners to register an interest in events of a specific type. The first pattern indicates that a Bean can multicast an event to multiple listeners. The second pattern indicates that a Bean can unicast an event to only one listener. The third pattern is used by a listener when it no longer wishes to receive a specific type of event notification from a Bean.
The following listing outlines a class that notifies other objects when a temperature value moves outside a specific range. The two methods indicated here allow other objects that implement the **TemperatureListener** interface to receive notifications when this occurs.

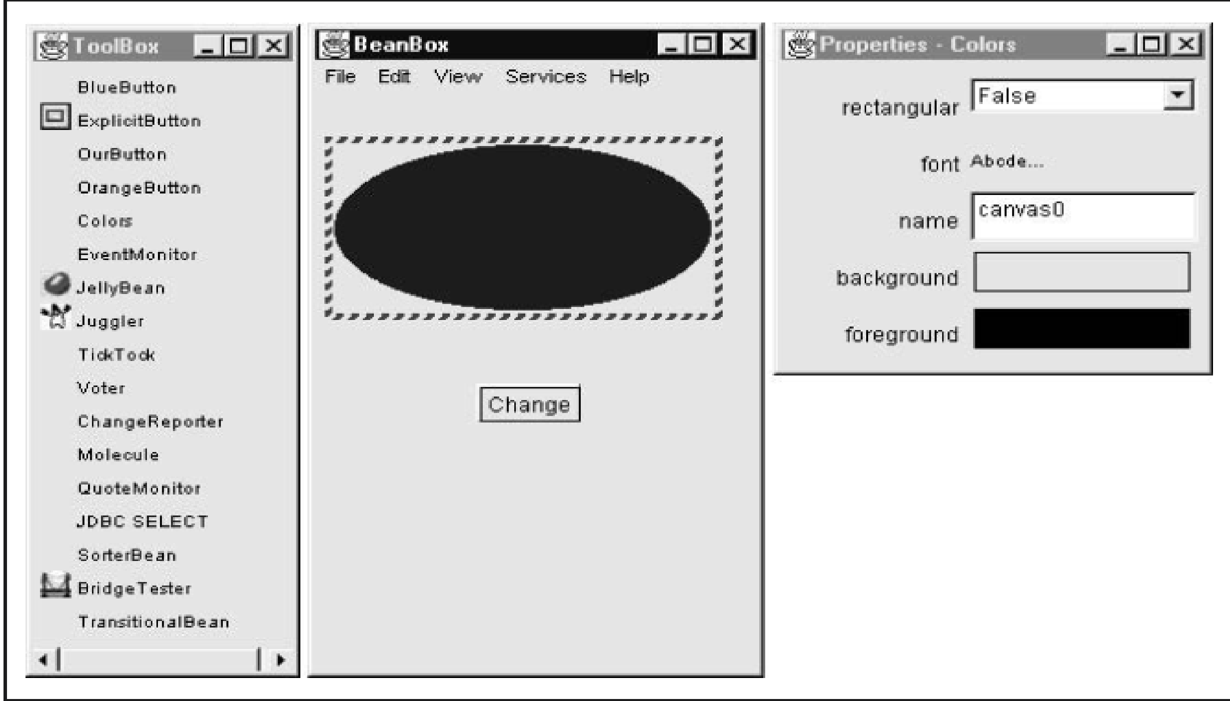
```
public class Thermometer {
public void addTemperatureListener(TemperatureListener tl) {
...
}
public void removeTemperatureListener(TemperatureListener tl) {
...
}
}
```

Methods

Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Developing a Simple Bean Using the BDK

The component is called the **Colors** Bean. It appears as either a rectangle or ellipse that is filled with a color. A color is chosen at random when the Bean begins execution. A public method can be invoked to change it. Each time the mouse is clicked on the Bean, another random color is chosen. There is one **boolean** read/write property that determines the shape.
The BDK is used to lay out an application with one instance of the **Colors** Bean and one instance of the **OurButton** Bean. The button is labeled “Change.” Each time it is pressed, the color changes.



Create a New Bean

1. Create a directory for the new Bean.
2. Create the Java source file(s).
3. Compile the source file(s).
4. Create a manifest file.
5. Generate a JAR file.
6. Start the BDK.
7. Test.

Create a Directory for the New Bean **c:\bdk\demo\sunw\demo\colors**. Then change to that directory.

Create the Source File for the New Bean

The source code for the **Colors** component is shown in the following listing. It is located in the file **Colors.java**.
The **import** statement at the beginning of the file places it in the package named **sunw.demo.colors**. this file must be located in a subdirectory named **sunw\demo\colors** relative to the **CLASSPATH** environment variable.
The color of the component is determined by the private **Color** variable **color**, and its shape is determined by the private **boolean** variable **rectangular**.

// A simple Bean.

```
package sunw.demo.colors;
import java.awt.*;
import java.awt.event.*;
public class Colors extends Canvas {
transient private Color color;
private boolean rectangular;
public Colors() {
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent me) {
change();
}
});
rectangular = false;
setSize(200, 100);
change();
}
public boolean getRectangular() {
return rectangular;
}
public void setRectangular(boolean flag) {
this.rectangular = flag;
repaint();
}
public void change() {
color = randomColor();
repaint();
}
private Color randomColor() {
int r = (int)(255*Math.random());
int g = (int)(255*Math.random());
int b = (int)(255*Math.random());
return new Color(r, g, b);
}
public void paint(Graphics g) {
Dimension d = getSize();
int h = d.height;
int w = d.width;
g.setColor(color);
if(rectangular) {
g.fillRect(0, 0, w-1, h-1);
}
else {
g.fillOval(0, 0, w-1, h-1);
}
}
}
```

Compile the Source Code for the New Bean

Compile the source code to create a class file. Type the following:

javac Colors.java.

Create a Manifest File

switch to the **c:\bdk\demo** directory. This is the directory in which the manifest files for the BDK demos are located. Put the source code for your manifest file in the file **colors.mft**.

```
Name: sunw/demo/colors/Colors.class
Java-Bean: True
```

This file indicates that there is one **.class** file in the JAR file and that it is a Java Bean. the **Colors.class** file is in the package **sunw.demo.colors** and in the subdirectory **sunw\demo\colors** relative to the current directory.

Generate a JAR File

Beans are included in the ToolBox window of the BDK only if they are in JAR files in the directory **c:\bdk\jars**. These files are generated with the jar utility. Enter the following:

jar cfm ..\jars\colors.jar colors.mft sunw\demo\colors*.class

This command creates the file **colors.jar** and places it in the directory **c:\bdk\jars**.

Start the BDK

Change to the directory **c:\bdk\beanbox** and type **run**. This causes the BDK to start.

The ToolBox window should include an entry labeled "Colors" for your new Bean. Create an Instance of the Colors Bean .Test your new component by pressing the mouse anywhere within its borders. Its color immediately changes.

Using the BeanInfo Interface

Design patterns were used to determine the information that was provided to a Bean user. This section describes how a developer can use the **BeanInfo** interface to explicitly control this process.

This interface defines several methods, including these:

```
PropertyDescriptor[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )
```

Constrained Properties

A Bean that has a *constrained* property generates an event when an attempt is made to change its value. The event is of type **PropertyChangeEvent**. It is sent to objects that previously registered an interest in receiving such notifications. Those other objects have the ability to veto the proposed change. This capability allows a Bean to operate differently according to its run-time environment. A full discussion of constrained

Persistence

Persistence is the ability to save a Bean to nonvolatile storage and retrieve it at a later time. The information that is particularly important are the configuration settings.

Customizers

The Properties window of the BDK allows a developer to modify the properties of a Bean. However, this may not be the best user interface for a complex component with many interrelated properties. Therefore, a Bean developer can provide a *customizer* that helps another developer configure this software. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided.

The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package. This section provides a brief overview of its contents. The Table lists the interfaces in **java.beans** and provides a brief description of their functionality.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets.
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred. (Added by Java 2, version 1.4.)
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.

Introduction to EJB 's

Enterprise JavaBean

Enterprise JavaBeans (EJB) is a comprehensive technology that provides the infrastructure for building enterprise-level server-side distributed Java components. The EJB technology provides a distributed component architecture that integrates several enterprise-level requirements such as distribution, transactions, security, messaging, persistence, and connectivity to mainframes and Enterprise Resource Planning (ERP) systems. When compared with other distributed component technologies such as Java RMI and CORBA, the EJB architecture hides most the underlying system-level semantics that are typical of distributed component applications, such as instance management, object pooling, multiple threading, and connection pooling. Secondly, unlike other component models, EJB technology provides us with different types of components for business logic, persistence, and enterprise messages.

Thus, an Enterprise Java Bean is a remote object with semantics specified for creation, invocation and deletion. The EJB container is assigned the system-level tasks mentioned above. What a web container does for Java servlets and JSPs in a web server, the EJB container is for EJBs.

EJB Architecture

Any distributed component technology should have the following requirements:

1. There should be a mechanism to create the client-side and server-side proxy objects. A client-side proxy represents the server-side object on the client-side. As far as the client is concerned, the client-side proxy is equivalent to the server-side object. On the other hand, the purpose of the server-side proxy is to provide the basic infrastructure to receive client requests and delegate these request to the actual implementation object

2. We need to obtain a reference to client-side proxy object. In order to communicate with the server-side object, the client needs to obtain a reference to the proxy.

3. There should be a way to inform the distributed component system that a specific component is no longer in use by the client.

In order to meet these requirements, the EJB architecture specifies two kinds of interfaces for each bean. They are home interface and remote interface. These interfaces specify the bean contract to the clients. However, a bean developer need not provide implementation for these interfaces. The home interface will contain methods to be used for creating remote objects. The remote interface should include business methods that a bean is able to serve to clients. One can consider using the home interface to specify a remote object capable of creating objects conforming to the remote interface. That is, a home interface is analogous to a factory of remote objects. These are regular Java interfaces extending the javax.ejb.EJBHome and javax.ejb.EJBObject interfaces respectively.

Types of EJBs

The EJB architecture is based on the concept that in an enterprise computing system, database persistence-related logic should be independent of the business logic that relies on the data. This happens to be a very useful technique for separating business logic concerns from database concerns. This makes that business logic can deal with the business data without worrying about how the data is stored in a relational database.

Enterprise JavaBeans server-side components come in two fundamentally different types: entity beans and session beans. Basically entity beans model business concepts that can be expressed as nouns. For example, an entity bean might represent a customer, a piece of equipment, an item in inventory. Thus entity beans model real-world objects. These objects are usually persistent records in some kind of database.

Session beans are for managing processes or tasks. A session bean is mainly for coordinating particular kinds of activities. That is, session beans are plain remote objects meant for abstracting business logic. The activity that a session bean represents is fundamentally transient. A session bean does not represent anything in a database, but it can access the database. Thus an entity bean has persistent state whereas a session bean models interactions but does not have persistent state.

Session beans are transaction-aware. In a distributed component environment, managing transactions across several components mandates distributed transaction processing. The EJB architecture allows the container to manage transactions declaratively. This mechanism lets a bean developer to specify transactions across bean methods. Session beans are client-specific. That is, session bean instances on the server side are specific to the client that created them on the client side. This eliminates the need for the developer to deal with multiple threading and concurrency.

Unlike session beans, entity beans have a client-independent identity. This is because an entity bean encapsulates persistent data. The EJB architecture lets a developer to register a primary key class to encapsulate the minimal set of attributes required to represent the identity of an entity bean. Clients can use these primary key objects to accomplish the database operations, such as create, locate, or delete entity beans. Since entity beans represent persistent state, entity beans can be shared across different clients. Similar to session beans, entity beans are also transactional, except for the fact that bean instances are not allowed to programmatically control transactions.

These two types of beans are meant for synchronous invocation. That is, when a client invokes a method on one of the above types, the client thread will be blocked till the EJB container completes executing the method on the bean instance. Also these beans are unable to service the messages which comes asynchronously over a messaging service such as JMS. To overcome this deficiency, the EJB architecture has introduced a third type of bean called message-driven bean. A message-driven bean is a bean instance that can listen to messages from the JMS.

Unlike other types of beans, a message-driven bean is a local object without home and remote interfaces. In a J2EE platform, message-driven beans are registered against JMS destinations. When a JMS message receives a destination, the EJB container invokes the associated message-driven bean. Thus message-driven beans do not require home and remote interfaces as instances of these beans are created based on receipt of JMS messages. This is an asynchronous activity and does not involve clients directly. The main purpose of message-driven beans is to implement business logic in response to JMS messages. For instance, take a B2B e-commerce application receiving a purchase order via a JMS message as an XML document. On receipt of such a message in order to persist this data and perform any business logic, one can implement a message-driven bean and associate it with the corresponding JMS destination. Also these beans are completely decoupled from the clients that send messages.

Session Beans: Stateful and Stateless

Session beans can be either stateful or stateless. Stateful session beans maintain conversational state when used by a client. Conversational state is not written to a database but can store some state in private variables during one method call and a subsequent method call can rely on this state. Maintaining a conversational state allows a client

to carry on a conversation with a bean. As each method on the bean is invoked, the state of the session bean may change and that change can affect subsequent method calls.

Stateless session beans do not maintain any conversational state. Each method is completely independent and uses only data passed in its parameters. One can specify whether a bean is stateful or not in the bean's deployment descriptor.

Entity Beans: Container and Bean Managed Persistence

An example entity bean in a B2B application is given as follows. A purchase order is a business identity and requires persistence store such as a relational database. The various purchase order attributes can be defined as the attributes of an entity bean. Since database operations involve create, update, load, delete, and find operations, the EJB architecture requires entity beans to implement these operations. Entity beans should implement the javax.ejb.EntityBean interface that specifies the load and delete operations among others. In addition, the bean developer should specify the appropriate create and find methods on the home interface, and provide their implementation in an entity bean.

There are two types of entity beans and they are distinguished by how they manage persistence. Container-managed beans have their persistence automatically managed by the EJB container. This is a more sophisticated approach and here the bean developer does not implement the persistence logic. The developer relies on the deployment descriptor to specify attributes whose persistence should be managed by the container. The container knows how a bean instance's fields map to the database and automatically takes care of inserting, updating, and deleting the data associated with entities in the database.

Beans using bean-managed persistence do all this work explicitly: the bean developer has to write the code to manipulate the database. The EJB container tells the bean instance when it is safe to insert, update, and delete its data from the database, but it provides no other help. The bean instance has to do the persistence work itself.

EJB Container: The environment that surrounds the beans on the EJB server is often referred to as the container. The container acts as an intermediary between the bean class and the EJB server. The container manages the EJB objects and EJB homes for a particular type of bean and helps these constructs to manage bean resources and apply the primary services relevant to distributed systems to bean instances at run time. An EJB server can have more than one container and each container in turn can accommodate more than one enterprise bean.

Remote Interface: This interface for an enterprise bean defines the enterprise bean's business methods that clients for this bean can access. The remote interface extends javax.ejb.EJBObject, which in turn extends java.rmi.Remote.

Home interface: This interface defines the bean's life cycle methods such as creation of new beans, removal of beans, and locating beans. The home interface extends javax.ejb.EJBHome, which in turn extends java.rmi.Remote.

Bean Class: This class has to implement the bean's business methods in the remote interface apart from some other callback methods. An entity bean must implement javax.ejb.EntityBean and a session bean must implement javax.ejb.SessionBean. Both EntityBean and Session Bean extend javax.ejb.EnterpriseBean.

Primary Key: This is a very simple class that provides a reference into the database. This class has to implement java.io.Serializable. Only entity beans need a primary key.

Deployment Descriptors: Much of the information about how beans are managed at runtime is not supplied in the interfaces and classes mentioned above. There are some common primary services related with distributed systems apart from some specific services such as security, transactions, naming that are being handled automatically by EJB server. But still EJB server needs to know beforehand how to apply the primary services to each bean class at runtime. Deployment descriptors exactly do this all important task.

JAR Files: Jar files are ZIP files that are used specifically for packaging Java classes that are ready to be used in some type of application. A Jar file containing one or more enterprise beans includes the bean classes, remote interfaces, home interfaces, and primary keys for each bean. It also contains one deployment descriptor.

Deployment is the process of reading the bean's JAR file, changing or adding properties to the deployment descriptor, mapping the bean to the database, defining access control in the security domain, and generating vendor-specific classes needed to support the bean in the EJB environment. Every EJB server product comes with its own deployment tools containing a graphical user interface and a set of command-line programs.

For clients like enterprise bean itself, Java RMI or CORBA client, to locate enterprise beans on the net, Java EJB specifications specify the clients to use Java Naming and Directory Interface (JNDI). JNDI is a standard Java extension that provides a uniform Application Programming Interface (API) for accessing a wide range of naming and directory services. The communication protocol may be Java RMI-IIOP or CORBA's IIOP.There are some special integrated application development tools such as Inprise's JBuilder, Sun's Forte and IBM's VisualAge, for designing EJBs in the market.

UNIT V : WEB SERVERS AND SERVLETS

CONTENTS

- Tomcat web server
- Introduction to servlets
- Life cycle of servelt
- Jsdk (java software development kit)
- The servlet API(application interface)
- The javax.servlet package
- Reading servlet parameters
- Reading initialization, parameters
- Javax.servlet http package
- Handling http request and response
- Using cookies-session tracking
- Security issues

TOMCAT WEB SERVER

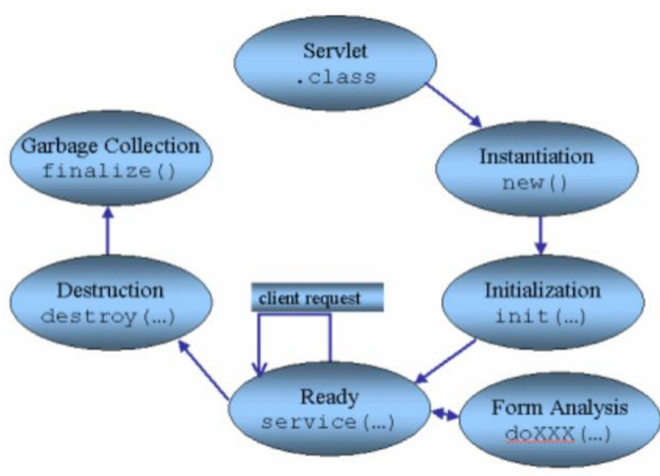
Apache Tomcat (or Jakarta Tomcat or simply Tomcat) is an open source servlet container developed by the Apache Software Foundation (ASF). Tomcat implements the Java Servlet and the JavaServer Pages (JSP) specifications from Oracle Corporation, and provides a "pure Java" HTTP web server environment for Java code to run. Apache Tomcat includes tools for configuration and management, but can also be configured by editing XML configuration files. Tomcat started off as a servlet reference implementation by James Duncan Davidson, a software architect at Sun Microsystems. He later helped make the project open source and played a key role in its donation by Sun to the Apache Software Foundation. The Apache Ant software build automation tool was developed as a side-effect of the creation of Tomcat as an open source project. Davidson had initially hoped that the project would become open sourced and, since many open source projects had O'Reilly books associated with them featuring an animal on the cover, he wanted to name the project after an animal. He came up with *Tomcat* since he reasoned the animal represented something that could fend for itself. Although the tomcat was already in use for another O'Reilly title, his wish to see an animal cover eventually came true when O'Reilly published their Tomcat book with a snow leopard on the cover.

INTRODUCTION TO SERVLETS

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases,or other software via the sockets and RMI mechanisms that you have seen already.

THE LIFE CYCLE OF A SERVLET

Three methods are central to the life cycle of a servlet. These are init( ), service( ), and destroy( ). They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called. First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server. Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server. Third, the server invokes the init( ) method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself. Fourth, the server invokes the service( ) method of the servlet. This method is called to process the HTTP request.



The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The service( ) method is called for each HTTP request. Finally, the server may decide to unload the servlet

from its memory. The algorithms by which this determination is made are specific to each server. The server calls the destroy( ) method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

JSDK : JAVA SOFTWARE DEVELOPMENT KIT

The JSWDK is the official reference implementation of the servlet 2.1 and JSP 1.0 specifications. It is used as a small stand-alone server for testing servlets and JSP pages before they are deployed to a full Web server that supports these technologies. It is free and reliable, but takes quite a bit of effort to install and configure.

install_dir/webpages/WEB-INF/servlets

Standard location for servlet classes.
• install_dir/classes
Alternate location for servlet classes.
• install_dir/lib
Location for JAR files containing classes.

THE SERVLET API

Two packages contain the classes and interfaces that are required to build servlets. These are javax.servlet and javax.servlet.http. They constitute the Servlet API. Keep in mind that these packages are not part of the Java core packages. Instead, they are standard extensions providedby Tomcat. Therefore, they are not included with Java SE 6. The Servlet API has been in a process of ongoing development and enhancement.

THE JAVAX.SERVLET PACKAGE

The javax.servlet package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes the core interfaces that are provided in this package. The most significant of these is Servlet. All servlets must implement this interface or extend a class that implements the interface. The ServletRequest and ServletResponse interfaces are also very important.

The Servlet Interface

All servlets must implement the Servlet interface. It declares the init( ), service( ), and destroy( ) methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters.

The init( ), service( ), and destroy( ) methods are the life cycle methods of the servlet. These are invoked by the server.

The ServletConfig Interface

The getServletConfig( ) method is called by the servlet to obtain initialization parameters. A servlet developer overrides the getServletInfo( ) method to provide a string with useful information (for example, author, version, date, copyright). This method is also invoked by the server.

The ServletContext Interface

The ServletContext interface enables servlets to obtain information about their environment.

The ServletRequest Interface
The ServletRequest interface enables a servlet to obtain information about a client request.

The ServletResponse Interface
The ServletResponse interface enables a servlet to formulate a response for a client.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Provides an input stream for reading requests from a client.
ServletOutputStream	Provides an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

Method	Description
ServletContext getServletContext( )	Returns the context for this servlet.
String getInitParameter(String *param*)	Returns the value of the initialization parameter named *param*.
Enumeration getInitParameterNames( )	Returns an enumeration of all initialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.

Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig sc) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from sc. An **UnavailableException** should be thrown if the servlet cannot be initialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from *req*. The response to the client can be written to *res*. An exception is generated if a servlet or IO problem occurs.

Method	Description
Object getAttribute(String *attr*)	Returns the value of the server attribute named *attr*.
String getMimeType(String *file*)	Returns the MIME type of *file*.
String getRealPath(String *vpath*)	Returns the real path that corresponds to the virtual path *vpath*.
String getServerInfo( )	Returns information about the server.
void log(String s)	Writes s to the servlet log.
void log(String s, Throwable *e*)	Writes s and the stack trace for *e* to the servlet log.
void setAttribute(String *attr*, Object *val*)	Sets the attribute specified by *attr* to the value passed in *val*.

Method	Description
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value –1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A **null** value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream** that can be used to read binary data from the request. An **IllegalStateException** is thrown if **getReader( )** has already been invoked for this request.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String *name*)	Returns an array containing values associated with the parameter specified by *name*.
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStream( )** has already been invoked for this request.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

The

Method	Description
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP DELETE request.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP GET request.
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP HEAD request.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP OPTIONS request.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP POST request.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP PUT request.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP TRACE request.
long getLastModified(HttpServletRequest *req*)	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

GenericServlet Class

The GenericServlet class provides implementations of the basic life cycle methods for a servlet. GenericServlet implements the Servlet and ServletConfig interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

void log(String *s*)
void log(String *s*, Throwable *e*)

Here, *s* is the string to be appended to the log, and *e* is an exception that occurred.

The ServletInputStream Class

The ServletInputStream class extends InputStream. It is implemented by the servlet container and provides an input stream that a servlet developer can use to read the data from a client request. It defines the default constructor. In addition, a method is provided to read bytes from the stream. It is shown here:

int readLine(byte[ ] *buffer*, int *offset*, int *size*) throws IOException

Here, *buffer* is the array into which *size* bytes are placed starting at *offset.* The method returns the actual number of bytes read or –1 if an end-of-stream condition is encountered.

The ServletOutputStream Class

The ServletOutputStream class extends OutputStream. It is implemented by the servlet container and provides an output stream that a servlet developer can use to write data to a client response. A default constructor is defined. It also defines the print( ) and println( ) methods, which output data to the stream.

The Servlet Exception Classes

javax.servlet defines two exceptions. The first is ServletException, which indicates that a servlet problem has occurred. The second is UnavailableException, which extends ServletException. It indicates that a servlet is unavailable.

<u>READING SERVLET PARAMETERS</u>

The ServletRequest interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in PostParameters.htm, and a servlet is defined in PostParametersServlet.java.

The HTMLsource code for PostParameters.htm is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"  method="post" action="http://localhost:8080/servlets-examples/
servlet/PostParametersServlet">
<table>
	<tr><td><B>Employee</td><td><input type=textbox name="e" size="25" value=""></td></tr>
	<tr><td><B>Phone</td><td><input type=textbox name="p" size="25" value=""></td> </tr>
</table>
```

<input type=submit value="Submit"> </body> </html>
The source code for PostParametersServlet.java is shown in the following listing. The service( ) method is overridden to process client requests. The getParameterNames( ) method returns an enumeration of the parameter names. These are processed in a loop. The parameter value is obtained via the getParameter( ) method.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet extends GenericServlet {

public void service(ServletRequest request,ServletResponse response) throws ServletException, IOException {
// Get print writer.
        PrintWriter pw = response.getWriter();
// Get enumeration of parameter names.
        Enumeration e = request.getParameterNames();
// Display parameter names and values.
        while(e.hasMoreElements()) {
                String pname = (String)e.nextElement();
                pw.print(pname + " = ");
                String pvalue = request.getParameter(pname);
                pw.println(pvalue);
        }
        pw.close();
        }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the web.xml file, perform these steps to test this example:

1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.
3. Enter an employee name and phone number in the text fields.
4. Submit the web page.

THE JAVAX.SERVLET.HTTP PACKAGE

The javax.servlet.http package contains a number of interfaces and classes that are commonly used by servlet developers. You will see that its functionality makes it easy to build servlets that work with HTTP requests and responses.
The following table summarizes the core interfaces that are provided in this package:

Interface	Description
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.
HttpSessionBindingListener	Informs an object that it is bound to or unbound from a session.

The following table summarizes the core classes that are provided in this package. The most important of these is HttpServlet. Servlet developers typically extend this class in order to process HTTP requests.

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.
HttpSessionEvent	Encapsulates a session-changed event.
HttpSessionBindingEvent	Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.

The HttpServletRequest Interface
The HttpServletRequest interface enables a servlet to obtain information about a client request.
The HttpServletResponse Interface
The HttpServletResponse interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, SC_OK indicates that the HTTP request succeeded, and SC_NOT_FOUND indicates that the requested resource is not available.
The HttpSession Interface
The HttpSession interface enables a servlet to read and write the state information that is associated with an HTTP session.. All of these methods throw an IllegalStateException if the session has already been invalidated.

USING COOKIES & SESSION TRACKING

The Cookie Class
The Cookie class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. A user visits an online store. Acookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store.
Aservlet can write a cookie to a user's machine via the addCookie( ) method of the HttpServletResponse interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.
The names and values of cookies are stored on the user's machine. Some of the information that is saved for each cookie includes the following:

• The name of the cookie
• The value of the cookie
• The expiration date of the cookie
• The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends. Otherwise, the cookie is saved in a file on the user's machine. The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the Web server.

There is one constructor for Cookie. It has the signature shown here:

Cookie(String *name*, String *value*)

Here, the name and value of the cookie are supplied as arguments to the constructor.

Method	Description
Object getAttribute(String *attr*)	Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found.
Enumeration getAttributeNames( )	Returns an enumeration of the attribute names associated with the session.
long getCreationTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
String getId( )	Returns the session ID.
long getLastAccessedTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request for this session.
void invalidate( )	Invalidates this session and removes it from the context.
boolean isNew( )	Returns **true** if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String *attr*)	Removes the attribute specified by *attr* from the session.
void setAttribute(String *attr*, Object *val*)	Associates the value passed in *val* with the attribute name passed in *attr*.

Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the maximum age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie is secure. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the version.
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int *v*)	Sets the version to *v*.

The HttpServlet Class
The HttpServlet class extends GenericServlet. It is commonly used when developing servlets that receive and process HTTP requests.
The HttpSessionEvent Class
HttpSessionEvent encapsulates session events. It extends EventObject and is generated when a change occurs to the session. It defines this constructor:

HttpSessionEvent(HttpSession *session*)
Here, *session* is the source of the event. HttpSessionEvent defines one method, getSession( ), which is shown here:
HttpSession getSession( ) : It returns the session in which the event occurred.

Example :

AddCookie.htm Allows a user to specify a value for the cookie named MyCookie.
AddCookieServlet.java Processes the submission of AddCookie.htm.
GetCookiesServlet.java Displays cookie values.

The HTML source code for AddCookie.htm is shown in the following listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to AddCookieServlet via an HTTP POST request.

```
<html>  <body> <center>
<form name="Form1"  method="post"
action="http://localhost:8080/servlets-examples/servlet/AddCookieServlet">
        <B>Enter a value for MyCookie:</B>
        <input type=textbox name="data" size=25 value="">
        <input type=submit value="Submit">
</form>  </body> </html>
```

The source code for AddCookieServlet.java gets the value of the parameter named “data”. It then creates a Cookie object that has the name “MyCookie” and contains the value of the “data” parameter. The cookie is then added to the header of the HTTP response via the addCookie( ) method. A feedback message is then
written to the browser.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet
{
public void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException,
IOException {
// Get parameter from HTTP request.
String data = request.getParameter("data");
// Create cookie.
Cookie cookie = new Cookie("MyCookie", data);
// Add cookie to HTTP response.
response.addCookie(cookie);
// Write output to browser.
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>MyCookie has been set to");
pw.println(data);
pw.close();
}
}
```

The source code for GetCookiesServlet.java invokes the getCookies( ) method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. getName( ) and getValue( ) methods are called to obtain this information.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Get cookies from header of HTTP request.
Cookie[] cookies = request.getCookies();
// Display these cookies.
response.setContentType("text/html");  PrintWriter pw = response.getWriter(); pw.println("<B>");

for(int i = 0; i < cookies.length; i++) {
                String name = cookies[i].getName();
                String value = cookies[i].getValue();
                pw.println("name = " + name +"; value = " + value);
        }
pw.close();
}
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the web.xml file. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display AddCookie.htm in a browser.
3. Enter a value for MyCookie.
4. Submit the web page.

Next, request the following URL via the browser:

http://localhost:8080/servlets-examples/servlet/GetCookiesServlet

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism. A session can be created via the getSession( ) method of HttpServletRequest. An HttpSession object is returned. This object can store a set of bindings that associate names with objects. The setAttribute( ), getAttribute( ), getAttributeNames( ), and removeAttribute( ) methods of HttpSession manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular client.

The following servlet illustrates how to use session state. The getSession( ) method gets the

current session. A new session is created if one does not already exist. The getAttribute( ) method is called to obtain the object that is bound to the name “date”. That object is a Date object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A Date object encapsulating the current date and time is then created. The setAttribute( ) method is called to bind the name “date” to this object.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet
{
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
                // Get the HttpSession object.
        HttpSession hs = request.getSession(true);
                // Get writer.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();  pw.print("<B>");
                // Display date/time of last access.
        Date date = (Date)hs.getAttribute("date");
        if(date != null) {
        pw.print("Last access: " + date + "<br>");
        }
                // Display current date/time.
date = new Date();
hs.setAttribute("date", date);
pw.println("Current date: " + date);
}
}
```

SECURITY ISSUES

security has become one of the most important topics in web programming. Security is the science of keeping sensitive information in the hands of authorized users. On the web, this boils down to three important issues:

Authentication : Being able to verify the identities of the parties involved
Confidentiality : Ensuring that only the parties involved can understand the communication
Integrity :Being able to verify that the content of the communication is not changed during transmission .

A client wants to be sure that it is talking to a legitimate server (authentication), and it also want to be sure that any information it transmits, such as credit card numbers, is not subject to eavesdropping (confidentiality). The server is also concerned with authentication and confidentiality. If a company is selling a service or providing sensitive information to its own employees, it has a vested interest in making sure that nobody but an authorized user can access it. And both sides need integrity to make sure that whatever information they send gets to the other party unaltered.

Authentication, confidentiality, and integrity are all linked by digital certificate technology. Digital certificates allow web servers and clients to use advanced cryptographic techniques to handle identification and encryption in a secure manner. Thanks to Java's built-in support for digital certificates, servlets are an excellent platform for deploying secure web applications that use digital certificate technology. We'll be taking a closer look at them later.

Security is also about making sure that crackers can't gain access to the sensitive data on your web server. Because Java was designed from the ground up as a secure, network-oriented language, it is possible to leverage the built-in security features and make sure that server add-ons from third parties are almost as safe as the ones you write yourself.

Running Servlets Securely

The Servlet Sandbox

Servlets built using JDK 1.1 generally operate with a security model called the "servlet sandbox." Under this model, servlets are either trusted and given open access to the server machine, or they're untrusted and have their access limited by a restrictive security manager. The model is very similar to the "applet sandbox," where untrusted applet code has limited access to the client machine.

A security manager is subclassed from java.lang.SecurityManager that is loaded by the Java environment to monitor all security-related operations: opening network connections, reading and writing files, exiting the program, and so on. Whenever an application, applet, or servlet performs an action that could cause a potential security breach, the environment queries the security manager to check its permissions. For a normal Java application, there is no security manager. When a web browser loads an untrusted applet over the network, however, it loads a very restrictive security manager before allowing the applet to execute.

Servlets can use the same technology, if the web server implements it. Local servlets can be trusted to run without a security manager, or with a fairly lenient one. For the Java Web Server 1.1, this is what happens when servlets are placed in the default servlet directory or another local source. Servlets loaded from a remote source, on the other hand, are by nature suspect and untrusted, so the Java Web Server forces them to run in a very restrictive environment where they can't access the local file system, establish network connections, and so on. All this logic is contained within the server and is invisible to the servlet, except that the servlet may see a SecurityException thrown when it tries to access a restricted resource. The servlet sandbox is a simple model, but it is already more potent than any other server extension technology to date.

UNIT VI: INTRODUCTION TO JSP (Java Server Pages)

CONTENTS

- The problem with servelt and introduction to jsp
- The anatomy of a JSP page
- JSP processing
- JSP Application Design with MVC setting up and JSP environment
- Installing the java software development kit
- Tomcat server installation
- Testing Tomcat.

THE PROBLEM WITH SERVLET AND INTRODUCTION TO JAVASERVERPAGES

JavaServer Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content. To simple write the regular HTML in the normal manner, using familiar Web-page-building tools. You then enclose the code for the dynamic parts in special tags, most of which start with <% and end with %>.

We can think of servlets as Java code with HTML inside; you can think of JSP as HTML with Java code inside. Now, neither servlets nor JSP pages are restricted to using HTML, but they usually do, and this over-simplified description is a common way to view the technologies. Now, despite the large apparent differences between JSP pages and servlets, behind the scenes they are the same thing. JSP pages are translated into servlets, the servlets are compiled, and at request time it is the compiled servlets that execute. So, writing JSP pages is really just another way of writing servlets.

Even though servlets and JSP pages are equivalent behind the scenes, they are not equally useful in all situations. Separating the static HTML from the dynamic content provides a number of benefits over servlets alone, and the approach used in JavaServer Pages offers several advantages over competing technologies.

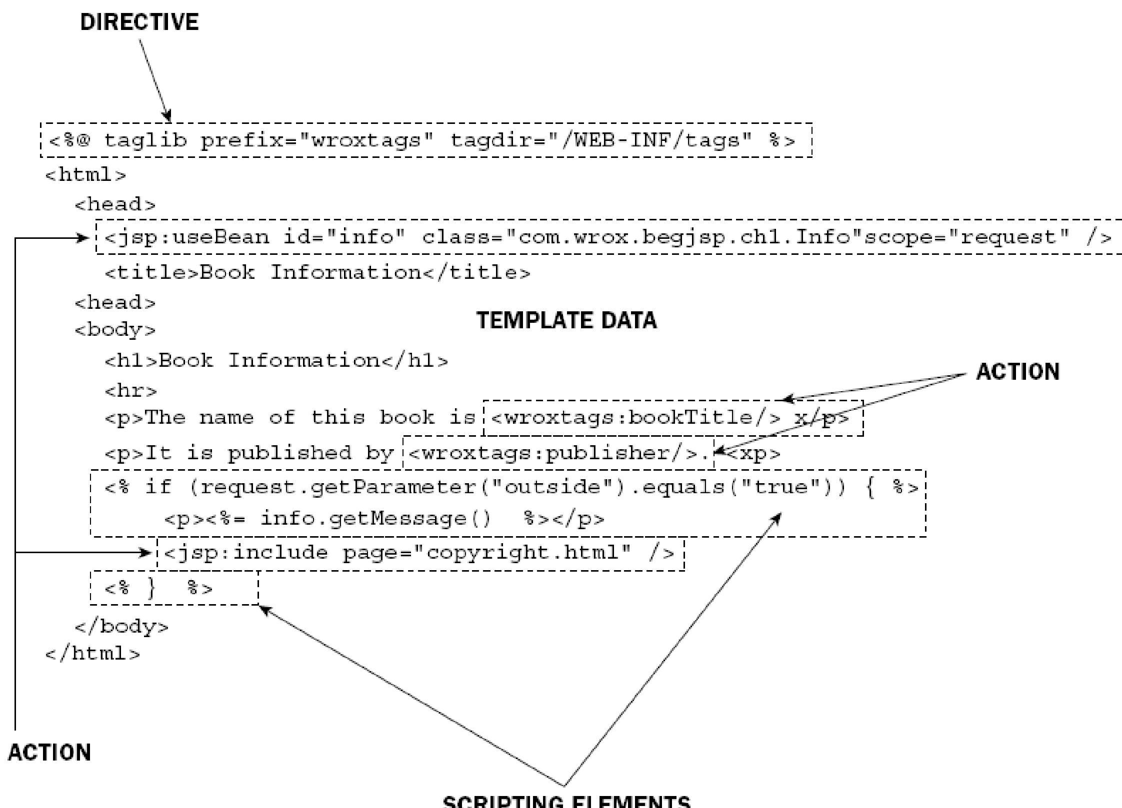
Benefits of JSP

JSP pages are translated into servlets. So, fundamentally, any task JSP pages can perform could also be accomplished by servlets. However, this underlying equivalence does not mean that servlets and JSP pages are equally appropriate in all scenarios. The issue is not the power of the technology, it is the convenience, productivity, and maintainability of one or the other. After all, anything you can do on a particular computer platform in the Java programming language you could also do in assembly language. But it still matters which you choose.

JSP provides the following benefits over servlets alone:

- It is easier to write and maintain the HTML. Your static code is ordinary HTML: no extra backslashes, no double quotes, and no lurking Java syntax.
- You can use standard Web-site development tools. For example, we use Macromedia Dreamweaver for most of the JSP pages in the book. Even HTML tools that know nothing about JSP can be used because they simply ignore the JSP tags.
- You can divide up your development team. The Java programmers can work on the dynamic code. The Web developers can concentrate on the presentation layer. On large projects, this division is very important. Depending on the size of your team and the complexity of your project, you can enforce a weaker or stronger separation between the static HTML and the dynamic content.

<u>**THE ANATOMY OF JSP**</u>



The visible elements that make up a JSP page can include the following:

- Directive elements
- Template data
- Action
- Scripting elements

A JSP page does not need to have all of these visible elements, but you will very likely encounter all of them if you look into any moderately complex JSP project. The following sections briefly describe each visible element.

Directives

Unlike other JSP elements, *directives* are not used to generate output directly. Rather, they are used to control some characteristics of a JSP page. Directives may be used to give special instructions to the JSP container about what to do during the translation of the page. You can always tell a directive from other elements because it is enclosed in a special set of braces: <%@ ... directive ... %>

Three directives are allowed in JSP:

❑ page directive ❑ taglib directive ❑ include directive

XML-compatible syntax

A directive — for example, a taglib directive — typically appears in a JSP as <%@ taglib .... %>
The same element can also appear in a JSP as : <jsp:directive.taglib .... />
This is an XML-compatible syntax for expressing the JSP directive. There are many advantages to expressing a JSP page in XML. For example, many developer tools can work with XML documents directly. Enterprise technologies, such as Web services, also make extensive use of XML. The emerging new XHTML standard is also XML-based. Because JSP works intimately with these technologies and the standard <%@ ... %> syntax is not valid XML, this alternative notation is necessary.
All professional JSP developers are trained in the <%@ ... %> notation. There are millions of lines of existing JSP code in this notation, so it will likely be supported for the foreseeable future.

Template data

Template data is static text. This static text is passed directly through the JSP container unprocessed. For example, it may be text that provides static HTML. The template data is the static HTML.
Although most JSP pages are used in generating HTML pages, JSP is not specific to HTML generation.
If a JSP page is expressed in XML syntax, typically contained in a .jspx file, the template data portion may have characters that need to be escaped. For example, the characters < and > are not allowed directly in an XML document and must be expressed as < and >, respectively.

Action

Action elements are JSP elements that are directly involved in the processing of the request. In most cases, action elements enable you to access data and manipulate or transform data in the generation of the dynamic output. For example, an online store may have a JSP page that displays a shopping cart. This cart JSP shows the products that you have purchased. Action elements may be used to generate the listing of the products (dynamic content) on the page and to calculate the cost and shipping (dynamic content), while template data (static HTML) is used to display the logo and shipping policy statements.
Action elements can be either *standard* or *custom*. A standard action is dependably available in every JSP container that conforms to the JSP 2.0 standard.
A custom action is an action created using JSP's tag extension mechanism. This mechanism enables developers to create their own set of actions for manipulating data or generating dynamic output within the JSP page.

Every XML tag has a name, optional attributes, and an optional body. For example, the standard

<jsp:include> action can be coded as follows:
<jsp:include page="news.jsp" flush="false"/>

The name of this tag is jsp:include, the attributes are page and flush, and this <jsp:include> instance does not have a body. The XML empty notation is used.
An XML tag can also have a body containing other tags, of course:
<jsp:include page="news.jsp" flush="false">
<jsp:param name="user" value="${param.username}"/> </jsp:include>

In this tag, the name is still jsp:include and the attributes are still page and flush, but now the body is no longer empty. Instead, the body contains a <jsp:param> standard action. After template data, actions are the next most frequently used elements in JSP coding. Actions are synonymous with tags because every action is an XML tag. The terms are used interchangeably in this book, just as they are in the JSP developer community.

Scripting elements

The practice of embedding code in another programming language within a JSP page is called *scripting*. Scripting elements are embedded code, typically in the Java programming language, within a JSP page. There are three different types of scripting elements:

❑ Declarations ❑ Scriptlets ❑ Expressions

Declarations are Java code that is used to declare variables and methods. They appear as follows:

<%! ... *Java declaration goes here*... %>

The XML-compatible syntax for declarations is:

<jsp:declaration> ... *Java declaration goes here* ... </jsp:declaration>

Scriptlets are arbitrary Java code segments. They appear as follows:

<% *... Java code goes here ...* %>

The XML-compatible syntax for scriptlets is:

<jsp:scriptlet> *... Java code goes here ...* </jsp:scriptlet>

Expressions are Java expressions that yield a resulting value. When the JSP is executed, this value is converted
to a text string and printed at the location of the scripting element. Expression scripting elements appear as: <%= ... Java expression goes here ... %>

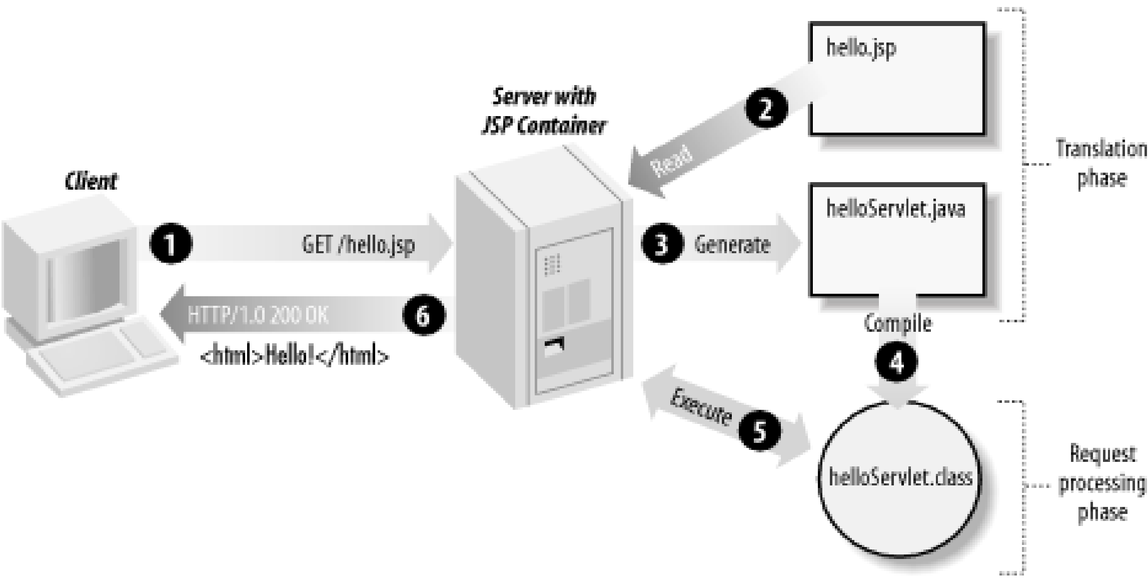
The XML-compatible syntax for expressions is:

<jsp:expression> ... Java expression goes here ... </jsp:expression>

JSP PROCESSING

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 responsible for intercepting requests for JSP pages. To process all JSP elements in the page, the container first turns the JSP page into a servlet (known as the *JSP page implementation class*). The conversion is pretty straightforward; all template text is converted to println( ) statements similar to the ones in the handcoded servlet and all JSP elements are converted to Java code that implements the corresponding dynamic behavior. The container then compiles the servlet class.

Converting the JSP page to a servlet and compiling the servlet form the *translation phase*. The JSP container initiates the translation phase for a page automatically when it receives the first request for the page. Since the translation phase takes a bit of time, the first user to request a JSP page notices a slight delay. The translation phase can also be initiated explicitly; this is referred to as *precompilation* of a JSP page. Precompiling a JSP page is a way to avoid hitting the first user with this delay. The JSP container is also responsible for invoking the JSP page implementation class (the generated servlet) to process each request and generate the response. This is called the *request processing phase*. The two phases are illustrated in *Figurebelow*



As long as the JSP page remains unchanged, any subsequent request goes straight to the request processing phase (i.e., the container simply executes the class file). When the JSP page is modified, it goes through the translation phase again before entering the request processing phase. 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 in one package under the name web container. A JSP page is really just another way to write a servlet without having to be a Java programming wiz. 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. By virtue of being an automatically generated servlet, a JSP page inherits all the advantages of a servlet as platform and vendor independence, integration, efficiency, scalability, robustness, and security.

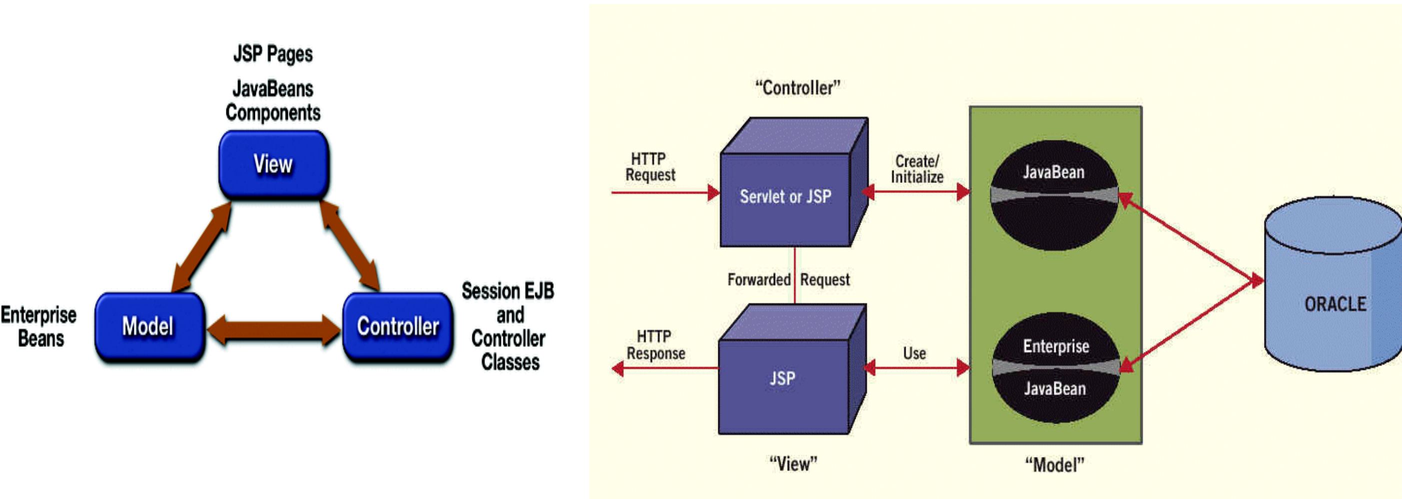
JSP APPLICATION DESIGN WITH MVC

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, I introduce a design model called Model-View-Controller (MVC), suitable for both simple and complex applications.

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 logic 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, such as customer, product, and order information. *Presentation* refers to how the application data is displayed to the user, for example, 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, and the Controller to the request processing.

MVC Architechture



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 many web sites go 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 the presentation makes it easier to evolve an application as the requirements change; new presentation interfaces can be developed without touching the business logic.

JSP pages are used as both the Controller and the View, and JavaBeans components are used as the Model. A single JSP page that handles everything, can use separate pages for the Controller and the View to make the application easier to maintain. Many types of real-world applications can be developed this way, but what's more important is that this approach allows you to examine all the JSP features without getting distracted by other technologies.

<u>INSTALLING JSDK (JAVA SOFTWARE DEVELOPMENT KIT)</u>

System Requirements

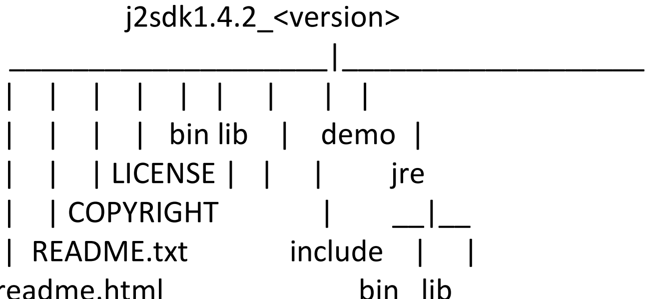
Software - Java 2 SDK Standard Edition, 1.4.2 is supported on i586 Intel and 100% compatible platforms running Microsoft Windows. For a list of supported operating systems and desktop managers.

Hardware - Intel and 100% compatible processors are supported. A Pentium 166MHz or faster processor with at least 32 megabytes of physical RAM is required to run graphically based applications. At least 48 megabytes of RAM is recommended for applets running within a browser using the Java Plug-in. Running with less memory may cause disk swapping which has a severe effect on performance. Very large programs may require more RAM for adequate performance.

Installation Instructions

In this procedure, we will run the self-installing executable to unpack and install the Java 2 SDK software bundle. As part of the Java 2 SDK, this installation includes the Java Plug-in and Java Web Start, as well as an option to include the public Java 2 Runtime Environment. The Java 2 SDK also contains a private J2RE for use only by its tools. For issues related to Windows Installation (IFTW) and Java Update, see the Windows Installation (IFTW) and Java Update FAQ. After the Java 2 SDK software has been installed, you may be asked to reboot your system.
<version> For example, if you are downloading the installer for update 1.4.2_01, the following file name: j2sdk-1_4_2_<version>-windows-i586.exe would become: j2sdk-1_4_2_01-windows-i586.exe
1. Check the download file size **(Optional)** If you save the self-installing executable to disk without running it from the download page at the web site, notice that its byte size is provided on the download page. Once the download has completed, check that you have downloaded the full, uncorrupted software file.
2. If 1.4.2 Beta is installed, uninstall it. Use the Microsoft Windows Add/Remove Programs utility, accessible from the Control Panel (Start -> Settings -> Control Panel).
3. Run the Java 2 SDK installer Note - you must have administrative permissions in order to install the Java 2 SDK on Microsoft Windows 2000 and XP.

Installed Directory Tree
The Java 2 SDK has the directory structure shown below.



TOMCAT SERVER & TESTING TOMCAT

Downloading and installing tomcat

The steps of downloading and installing are easy and you can learn the process very fast. Click the link: http://tomcat.apache.org/download-60.cgi and follow the steps according to your requirement to achieve Tomcat server.

Step 1:

Installation of JDK:

Before beginning the process of installing Tomcat on your system, ensure first the availability of JDK on your system program directory. Install it on your system if not already installed (because any version of tomcat requires the Java 1.6 or higher versions) and then set the class path (environment variable) of JDK. To set the **JAVA_HOME Variable:** you need to specify the location of the java run time environment to support the Tomcat else Tomcat server can not run. This variable contains the path of JDK installation directory.

```
set JAVA_HOME=C:\Program Files\Java\jdk1.6
```

For Windows OS, go through the following steps:

```
Start menu->Control Panel->System->Advanced tab-Environment Variables->New->set the Variable name = JAVA_HOME and variable value = C:\Program Files\Java\jdk1.6
```

Now click on all the subsequent ok buttons one by one. It will set the JDK path.

Step 2:

For setting the class path variable for JDK, do like this:

```
Start menu->Control Panel->System->Advanced tab->Environment Variables->New->
Set PATH="C:\Program Files\Java\jdk1.6\bin"; %PATH%
```

OR

First, right click on the

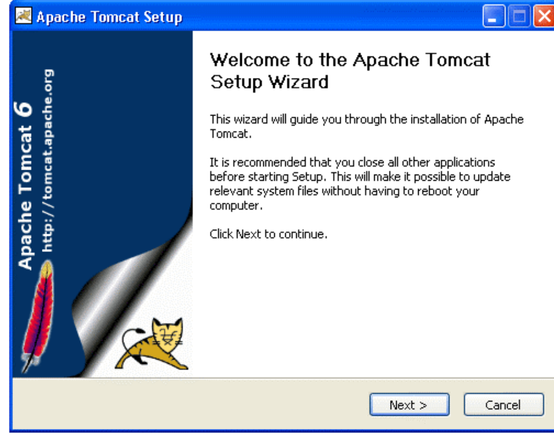
```
My Computer->properties->advance->Environment Variables->path.
```

Now, set bin directory path of JDK in the path variable

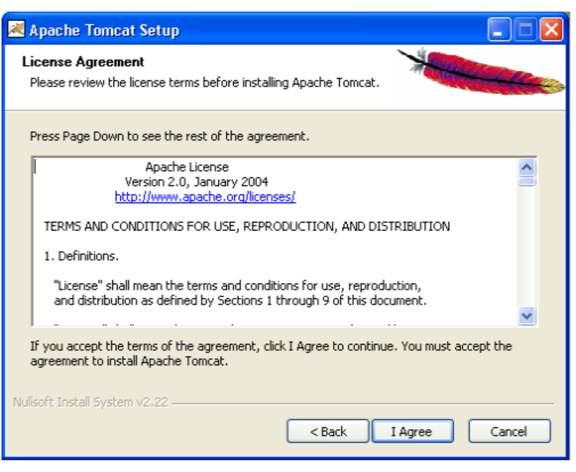
Step 3:

The process of installing Tomcat 6.0 begins here from now. It takes various steps for installing and configuring the Tomcat 6.0. For Windows OS, Tomcat comes in two forms: .zip file and .exe file (the Windows installer file). Here we are exploring the installation process by using the .exe file. First unpack the zipped file and simply execute the '.exe' file.

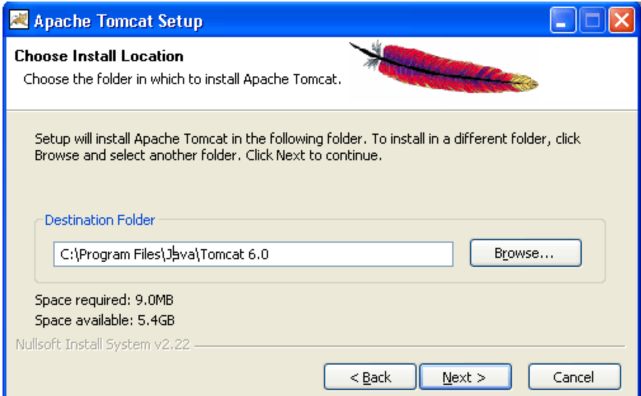
A Welcome screen shot appears that shows the beginning of installation process. Just click on the 'Next' button to proceed the installation process.



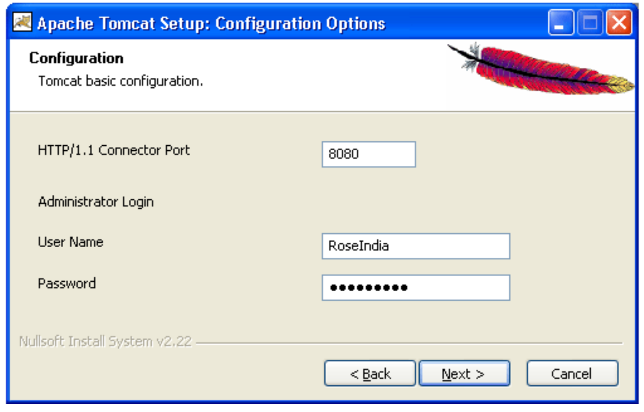
Steps 4:
A screen of 'License Agreement' displays.
Click on the 'I Agree' button.



Step 5:
A screen shot appears asking for the 'installing location' choose the default components and click on the 'Next' button.

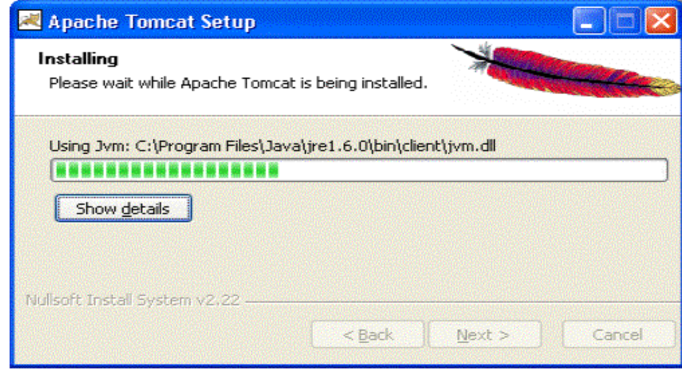


Step 6:
A screen shot of 'Configuration Options' displays on the screen. Choose the location for the Tomcat files as per your convenience. You can also opt the default Location The port number will be your choice on which you want to run the tomcat server. The port number 8080 is the default port value for tomcat server to proceed the HTTP requests. The user can also change the 'port number' after completing the process of installation; for this, users have to follow the following tips. Go to the specified location as " **Tomcat 6.0 \conf \server.xm**l ". Within the server.xml file choose "Connector" tag and change the port number. Now, click on the 'Next' button to further proceed the installation process.

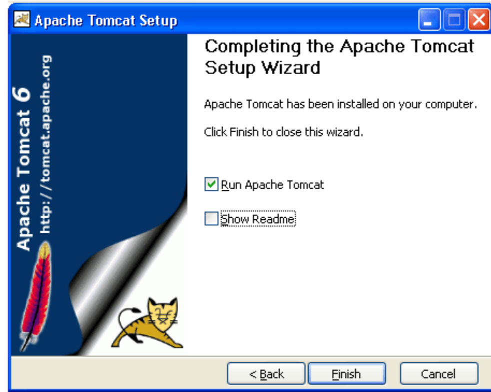


Step 7:
A Window of Java Virtual Machine displays on the screen .This window asks for the location of the installed Java Virtual Machine. Browse the location of the JRE folder and click on the Install button. This will install the Apache tomcat at the specified location.

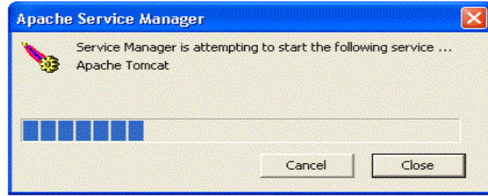
Step 8:
A processing window of installing displays on the screen. To get the information about installer click on the "Show details" button



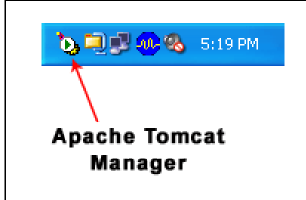
Step 9:
A screen shot of 'Tomcat Completion' displays on the screen. Click on the 'Finish' button.



Step 10:
A window of Apache Service Manager appears with displaying the running process.Let the running process goes on.

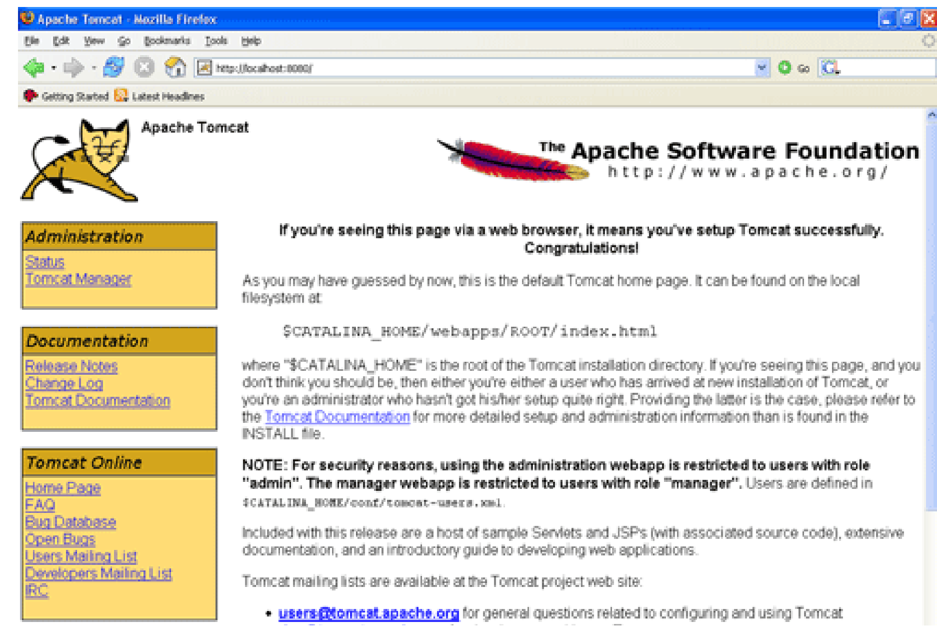


Step 11:
After completing the installation process, the Apache Tomcat Manager appears on the toolbar panel like shown in the picture.



Start the Tomcat Server:

1.Start the tomcat server from the bin folder of Tomcat 6.0 directory by double clicking the "tomcat6.exe" file.
OR create a shortcut of this .exe file at your desktop.
2. Now Open web browser and type URL http://localhost:8080 in the address bar to test the server
3. To Stop the Tomcat Server: Stop the server by pressing the "Ctrl + c" keys.
The screen of Apache Tomcat software looks like this:


Apache Tomcat - Mozilla Firefox
Apache Tomcat
The Apache Software Foundation
http://www.apache.org/
Administration
Status
Tomcat Manager
Documentation
Release Notes
Change Log
Tomcat Documentation
Tomcat Online
Home Page
FAQ
Bug Database
Open Bugs
Users Mailing List
Developers Mailing List
IRC
If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!
As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:
$CATALINA_HOME/webapps/ROOT/index.html
where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.
NOTE: For security reasons, using the administration webapp is restricted to users with role "admin". The manager webapp is restricted to users with role "manager". Users are defined in $CATALINA_HOME/conf/tomcat-users.xml.
Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation, and an introductory guide to developing web applications.
Tomcat mailing lists are available at the Tomcat project web site:
users@tomcat.apache.org for general questions related to configuring and using Tomcat

UNIT VII: JSP APPLICATION DEVELOPMENT

CONTENTS

- Generic dynamic content
- Using scripting elements
- Implicit jsp objects
- Conditional processing
- Using get and set attributes
- Declaring variables
- Method error handling and debugging sharing data between jsp pagesrequest
- Request and user passing control and data between the pages
- Sharing session and application data
- Memory usage considerations

GENERATING DYNAMIC CONTENT

In this chapter, we develop a page for displaying the current date and time, and look at the JSP directive element and how to use JavaBeans in a JSP page along the way. Next, we look at how to process user input in your JSP pages and make sure it has the appropriate format. Recall from JSP pages should have the file extension .jsp, which tells the server that the page needs to be processed by the JSP container. Without this clue, the server is unable to distinguish a JSP page from any other type of file and sends it unprocessed to the browser.The first example JSP page, named date.jsp

Example JSP Page Showing the Current Date and Time (date.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
  <body bgcolor="white">
    <jsp:useBean id="clock" class="java.util.Date" />
    The current time at the server is:
   <ul>      <li>Date: <jsp:getProperty name="clock" property="date" />
     <li>Month: <jsp:getProperty name="clock" property="month" />
     <li>Year: <jsp:getProperty name="clock" property="year" />
     <li>Hours: <jsp:getProperty name="clock" property="hours" />
     <li>Minutes: <jsp:getProperty name="clock" property="minutes" />
   </ul>   </body>  </html>
```

The date.jsp page displays the current date and time. Setting Up the JSP Environment, first start the Tomcat server and load the http://localhost:8080/ora/ URL in a browser.

Figure Current Date/Time JSP page example

This page contains both regular HTML elements and JSP elements. The HTML elements are used as-is, defining the layout of the page. If you use the View Source function in your browser, you notice that none of the JSP elements are visible in the page source. That's because the JSP elements are processed by the server when the page is requested, and only the resulting output is sent to the browser. To see the unprocessed JSP page in a separate window, click on the source link for the date.jsp file in the book

examples main page. The source link uses a special servlet to send the JSP page as-is to the browser instead of letting the server process it. This makes it easier for you to compare the source page and the processed result.

USING SCRIPTING ELEMENTS

There are three types of JSP elements you can use: directive, action, and scripting.

Directive elements

The directive elements, shown in Table , specify information about the page itself that remains the same between requests--for example, if session tracking is required or not, buffering requirements, and the name of a page that should be used to report errors, if any.

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

Standard action elements

Action elements typically perform some action based on information that is required at the exact time the JSP page is requested by a browser. An action 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

Action 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 component 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 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 browser-dependent elements (OBJECT or EMBED) needed to execute an applet with the Java Plug-in software

Custom action elements and the JSP Standard Tag Library

In addition to the standard actions, the JSP specification includes a Java API a programmer can use to develop custom actions to extend the JSP language. The JSP Standard Tag Library (JSTL) is such an extension, with the special status of being defined by a formal specification from Sun and typically bundled with the JSP container. JSTL contains action elements for processes needed in most JSP applications, such as conditional processing, database access, internationalization, and more.

Scripting elements

Scripting elements allow to add small pieces of code (typically Java code) in 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.

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

JavaBeans components

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 components are typically used as containers for information that describes application entities, such as a customer or an order.

Using JSP Directives

Directives are used to specify attributes of the page itself, primarily those that affect how the page is converted into a Java servlet. There are three JSP directives: page, include, and taglib. In this example, we're using only the page directive. JSP pages typically start with a page directive that specifies the scripting language and the content type for the page: <%@ page language="java" contentType="text/html" %> A JSP directive element starts with a directive-start identifier (<%@) followed by the directive name (e.g., page) and directive attributes, and ends with %>. A directive contains one or more attribute name/value pairs (e.g., language="java"). Note that JSP element and attribute names are case-sensitive, and in most cases the same is true for attribute values. For instance, the language attribute value must be java, not Java. All attribute values must also be enclosed in single or double quotes.

The language attribute specifies the scripting language used in the page. The JSP reference implementation (the Tomcat server) supports only Java as a scripting language. java is also the default value for the language attribute, but for clarity you may still want to specify it. Other JSP implementations support other languages besides Java, and hence allow other values for the language attribute.

Using JavaBeans

There is also some dynamic content in this example. Step back a moment and think about the type of dynamic content you see on the Web every day. Common examples might be a list of web sites matching a search criteria on a search engine site, the content of a shopping cart on an e-commerce site, a personalized news page, or messages on a bulletin board. Dynamic content is content generated by some

server process, for instance the result of a database query. Before it is sent to the browser, the dynamic content needs to be combined with regular HTML elements into a page with the right layout, navigation bars, the company logo, and so forth. In a JSP page, the regular HTML is the template text described earlier. The result of the server processing--the dynamic content--is commonly represented by a JavaBeans component.

A JavaBeans component, or just a bean for short, is a Java class that follows certain coding conventions, so it can be used by tools as a component in a larger application. In this chapter, we discuss only how to use a bean, not how to develop one. A bean is often used in JSP as the container for the dynamic content to be displayed by a web page. Typically, a bean represents something specific, such as a person, a product, or a shopping order. A bean is always created by a server process and given to the JSP page. The page then uses JSP elements to insert the bean's data into the HTML template text. The type of element used to access a bean in a page is called a JSP action element. JSP action elements are executed when a JSP page is requested (this is called the request processing phase, as you may recall from Chapter 3). In other words, JSP actions represent dynamic actions that take place at runtime, as opposed to JSP directives, which are used only during the translation phase (when the JSP page is turned into Java servlet code). JSP defines a number of standard actions and also specifies how you can develop custom actions. For both standard and custom action elements, use the following notation:

<action_name attr1="value1" attr2="value2"> action_body </action_name>

Action elements, or tags as they are sometimes called and are grouped into libraries (known as tag libraries). The action name is composed of two parts: a library prefix and the name of the action within the library, separated by a colon (i.e., jsp:useBean). All actions in the JSP standard library use the prefix jsp, while custom actions can use any prefix except jsp, jspx, java, javax, servlet, sun, or sunw. You specify input to the action through attribute/value pairs in the opening tag. The attribute names are case-sensitive, and the values must be enclosed in single or double quotes. For some actions, you can also enter data that the action should process in the action's body. It can be any text value, such as a SQL statement, or even other nested JSP action elements.

<jsp:useBean id="clock" class="java.util.Date" />

The id attribute is used to give the bean a unique name. It must be a name that is a valid Java variable name: it must start with a letter and cannot contain special characters such as dots, plus signs, etc. The class attribute contains the fully qualified name of the bean's Java class. Here, the name clock is associated with an instance of the class java.util.Date. Note that we don't specify a body for this action. When you omit the body, you must end the opening tag with />, as in this example. In this case, when the JSP container encounters this directive, there is no bean currently available with the name clock, so the <jsp:useBean> action creates a bean as an instance of the specified class and makes it available to other actions in the same page. In Chapter 8, Sharing Data Between JSP Pages, Requests, and Users, you will see how <jsp:useBean> can also be used to locate a bean that has already been created. Incidentally, the <jsp:useBean> action supports three additional attributes: scope, type, and beanName. The scope attribute is described in detail in Chapter 8, and the other two attributes are covered in Appendix A, JSP Elements Syntax Reference. We don't need to worry about those attributes here.

Accessing JavaBean Properties

The bean's data is represented by its properties. If you're a page author charged with developing a JSP page to display the content represented by a bean, you first need to know the names of all the bean's properties. This information should be available from the Java programmers on the team or from a third-party source. In this example, we use a standard Java class named java.util.Date as a bean with properties representing date and time information. Table describes the properties of date.

Table : Properties for java.util.Date

Property Name	Java Type	Access	Description
date	int	read	The day of the month as a number between 1 and 31
hours	int	read	The hour as a number between 0 (midnight) and 23
minutes	int	read	The number of minutes past the hour as a number between 0 and 59
month	int	read	The month as a number from 0 to 11
year	int	read	The current year minus 1900

Once you have created a bean and given it a name, you can retrieve the values of the bean's properties in the response page with another JSP standard action, <jsp:getProperty>. This action obtains the current value of a bean property and inserts it directly into the response body.

To include the current date property value in the page, use the following tag:

```
<jsp:getProperty name="clock" property="date" />
```

The name attribute, set to clock, refers to the specific bean instance we defined with the <jsp:useBean> action previously. This action locates the bean and asks it for the value of the property specified by the property attribute. As documented in the date property contains the day of the month as a number between 1 and 31. In multiple <jsp:getProperty> actions are used to generate a list of all the clock bean's property values.

Input and Output

User input is a necessity in modern web pages. Most dynamic web sites generate pages based on user input. Unfortunately, users seldom enter information in exactly the format you need, so before you can use such input, you probably want to validate it.

Using JavaBeans to Process Input

A bean is often used as a container for data, created by some server process, and used in a JSP page that displays the data. But a bean can also be used to capture user input. The captured data can then be processed by the bean itself or used as input to some other server component (e.g., a component that stores the data in a database or picks an appropriate banner ad to display). The nice thing about using a bean this way is that all information is in one bundle. Say you have a bean that can contain information about a person, and it captures the name, birth date, and email address as entered by the person on a

web form. You can then pass this bean to another component, providing all the information about the user in one shot. Now, if you want to add more information about the user, you just add properties to the bean, instead of having to add parameters all over the place in your code. Another benefit of using a bean to capture user input is that the bean can encapsulate all the rules about its properties. Thus, a bean representing a person can make sure the birthDate property is set to a valid date.

Setting JavaBeans properties from user input

Table Properties for com.ora.jsp.beans.userinfo.UserInfoBean

Property Name	Java Type	Access	Description
userName	String	read/write	The user's full name
birthDate	String	read/write	The user's birth date in the format yyyy-mm-dd (e.g., 2000-07-07)
emailAddr	String	read/write	The user's email address in the format name@company.com
sex	String	read/write	The user's sex (male or female)
luckyNumber	String	read/write	The user's lucky number (between 1 and 100)
valid	boolean	read	true if the current values of all properties are valid, false otherwise

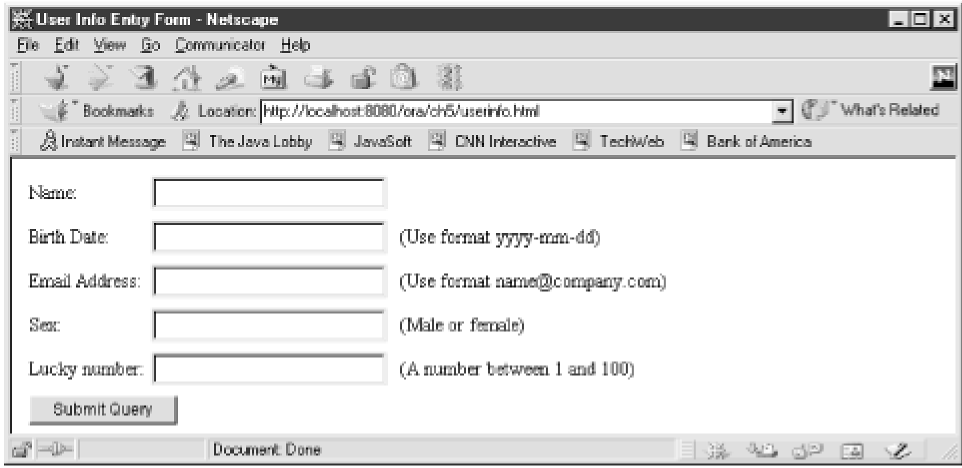
Example to capture data using html form : An HTML Form that Sends User Input to a JSP Page (userinfo.html)

```
<html>   <head>    <title>User Info Entry Form</title>   </head>
  <body bgcolor="white">
    <form action="userinfo1.jsp" method="post">
      <table>
        <tr>        <td>Name:</td>        <td><input type="text" name="userName" >        </td>
        </tr>
        <tr>        <td>Birth Date:</td>        <td><input type="text" name="birthDate" >        </td>
          <td>(Use format yyyy-mm-dd)</td>      </tr>
        <tr>
          <td>Email Address:</td>        <td><input type="text" name="emailAddr" >
          </td>        <td>(Use format name@company.com)</td>       </tr>
        <tr>
          <td>Sex:</td>        <td><input type="text" name="sex" >
          </td>
          <td>(Male or female)</td>
        </tr>
        <tr>
          <td>Lucky number:</td>
          <td><input type="text" name="luckyNumber" >
          </td>
          <td>(A number between 1 and 100)</td>
```

```
      </tr>
      <tr>
        <td colspan=2><input type="submit"></td>
      </tr>     </table>   </form>   </body> </html>
```

This is a regular HTML page that presents a form with a number of fields, as shown in Figure 5-3. There are a few things worth mentioning here. First, notice that each input field has a name attribute with a value that corresponds to a UserInfoBean property name. Matching the names lets us take advantage of a nice JSP feature that sets property values automatically, as you'll see shortly. Also note that the action attribute of the form specifies that a JSP page, userinfo1.jsp, is invoked when the user clicks the Submit button.

Figure . User input form



Example : A JSP Page that Validates User Input with a Bean (userinfo1.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
  <body bgcolor="white">
    <jsp:useBean     id="userInfo"     class="com.ora.jsp.beans.userinfo.UserInfoBean">
      <jsp:setProperty name="userInfo" property="*" />
    </jsp:useBean>
     The following information was saved:
    <ul>
      <li>User Name: <jsp:getProperty
                name="userInfo" property="userName" />
      <li>Birth Date: <jsp:getProperty
                 name="userInfo" property="birthDate" />
      <li>Email Address: <jsp:getProperty
                   name="userInfo" property="emailAddr" />
      <li>Sex: <jsp:getProperty
            name="userInfo" property="sex" />
      <li>Lucky number: <jsp:getProperty
```

```
                    name="userInfo" property="luckyNumber" />
    </ul>
    The user input is valid: <jsp:getProperty
                       name="userInfo" property="valid" />
  </body>
</html>
<jsp:useBean   id="userInfo"   class="com.ora.jsp.beans.userinfo.UserInfoBean">
  <jsp:setProperty name="userInfo" property="*" />
</jsp:useBean>
```

When a form is submitted, the form field values are sent as request parameters with the same names as the form field elements. note that an asterisk (*) is used as the property attribute value of the <jsp:setProperty> action. This means that all bean properties with names that match request parameters sent to the page are set automatically. That's why it's important that the form element names match the bean property names, as they do here. Automatically setting all matching properties is a great feature; if you define more properties for your bean, you can set them simply by adding new matching fields in the form that invokes the JSP page.
Besides the property attribute, the <jsp:setProperty> action has two more optional attributes: param and value. If for some reason you can't use the same name for the parameters and the property names, you can use the param attribute to set a bean property to the value of any request parameter:

```
<jsp:setProperty   name="userInfo"  property="userName"  param="someOtherParam"/>
```

Here, the userName property is set to the value of a request parameter named someOtherParam.
You can also explicitly set a bean property to a value that is not sent as a request parameter with the value attribute:

```
<jsp:setProperty   name="userInfo"  property="luckyNumber"  value="13"/>
```

Here, the luckyNumber property is set to the value 13. You typically use the value attribute only when you set the bean properties based on something other than user input, for instance values collected from a database.

JSP IMPLICIT OBJECTS

Implicit objects in jsp are the objects that are created by the container automatically and the container makes them available to the developers, the developer do not need to create them explicitly. Since these objects are created automatically by the container and are accessed using standard variables; hence, they are called implicit objects. The implicit objects are parsed by the container and inserted into the generated servlet code. They are available only within the jspService method and not in any declaration. Implicit objects are used for different purposes. Our own methods (user defined methods) can't access them as they are local to the service method and are created at the conversion time of a jsp into a servlet. But we can pass them to our own method if we wish to use them locally in those functions.

There are nine implicit objects. Here is the list of all the implicit objects:

Object	Class
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
exception	java.lang.Throwable
out	javax.servlet.jsp.JspWriter
page	java.lang.Object
PageContext	javax.servlet.jsp.PageContext
request	javax.servlet.ServletRequest
response	javax.servlet.ServletResponse
session	javax.servlet.http.HttpSession

- **Application:** These objects has an application scope. These objects are available at the widest context level, that allows to share the same information between the JSP page's servlet and any Web components with in the same application.
- **Config:** These object has a page scope and is an instance of javax.servlet.ServletConfig class. Config object allows to pass the initialization data to a JSP page's servlet. Parameters of this objects can be set in the deployment descriptor (web.xml) inside the element <jsp-file>. The method getInitParameter() is used to access the initialization parameters.
- **Exception:** This object has a page scope and is an instance of java.lang.Throwable class. This object allows the exception data to be accessed only by designated JSP "error pages."
- **Out:** This object allows us to access the servlet's output stream and has a page scope. Out object is an instance of javax.servlet.jsp.JspWriter class. It provides the output stream that enable access to the servlet's output stream.
- **Page:** This object has a page scope and is an instance of the JSP page's servlet class that processes the current request. Page object represents the current page that is used to call the methods defined by the translated servlet class. First type cast the servlet before accessing any method of the servlet through the page.
- **Pagecontext:** PageContext has a page scope. Pagecontext is the context for the JSP page itself that provides a single API to manage the various scoped attributes. This API is extensively used if we are implementing JSP custom tag handlers. PageContext also provides access to several page attributes like including some static or dynamic resource.
- **Request:** Request object has a request scope that is used to access the HTTP request data, and also provides a context to associate the request-specific data. Request object implements javax.servlet.ServletRequest interface. It uses the getParameter() method to access the request parameter. The container passes this object to the _jspService() method.
- **Response:** This object has a page scope that allows direct access to the **HTTPServletResponse** class object. Response object is an instance of the classes that implements the javax.servlet.ServletResponse class. Container generates to this object and passes to the _jspService() method as a parameter.
- **Session:** Session object has a session scope that is an instance of javax.servlet.http.HttpSession class. Perhaps it is the most commonly used object to manage the state contexts. This object persist information across multiple user connection.

JSP ERROR HANDLING

An exception is an event that occurs during the execution of a program and it disrupts the normal working of the instructions in the program.

Exception Handling in JSP

An exception is an event that occurs during the execution of a program and it disrupts the normal working of the instructions in the program. An exception can occur if you trying to connect to a database which doesn't exists or the database server is down, it may be thrown if you are requesting for a file which is unavailable, then the exception will be thrown to you. We can handle the exceptions in jsp by specifying errorPage in the page directive, <%@ page errorPage = "errorPage.jsp">. If any exception will be thrown then the control to handle the exception will be passed to that error page where we can display a information to the user about what's the reason behind throwing the exception. In this example we are going to handle the run- time exception. To make a program on this we are using three pages.

A Html Form: It is used to display a form to the user where he will enter the first and second number.
Controller class: This class is a jsp page which recieves the values entered by the user and prints it on the user screen. If any exceptions exception occurs then it will forward it other page which will handle the exception.
Exceptional Handler: This jsp page is actually an error page to which the control will be passed by the controller when the exception is thrown.

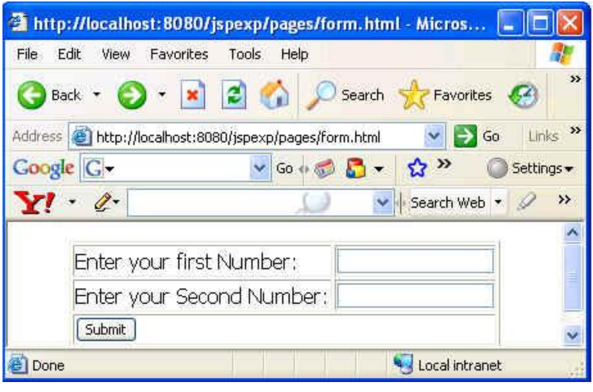
Code of the program is given below:

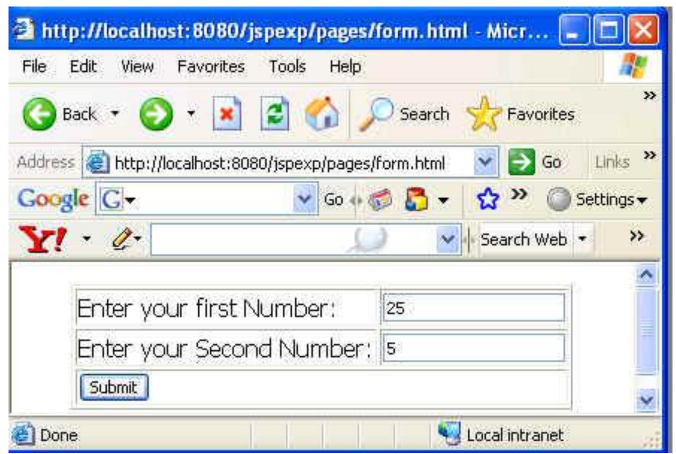
```
<html>
<head>
        <style>
        body, input { font-family:Tahoma; font-size:8pt; }
        </style>
</head>
<body>
<table align="center" border=1>
<form action="formHandler.jsp" method="post">
        <tr><td>Enter your first Number: </td>
        <td><input type="text" name="fno" /></td></tr>
        <tr><td>Enter your Second Number: </td>
        <td><input type="text" name="sno" /></td></tr>
        <tr ><td colspan="2"><input ty
  pe="submit" value="Submit" /></td></tr>
</form>
</table>
</body>
</html>
```

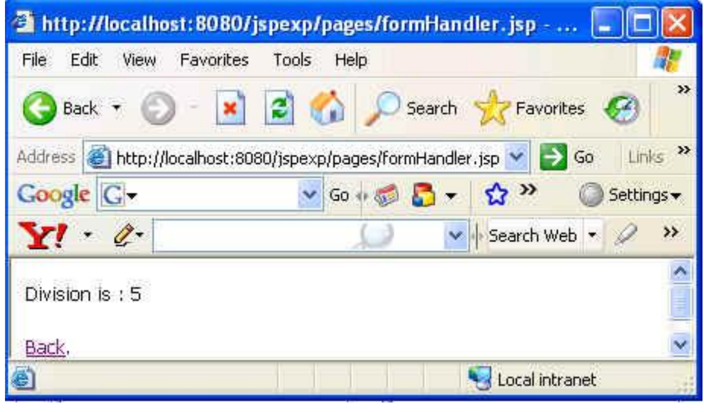
```
<%@ page errorPage="exceptionHandler.jsp" %>
<html>
```

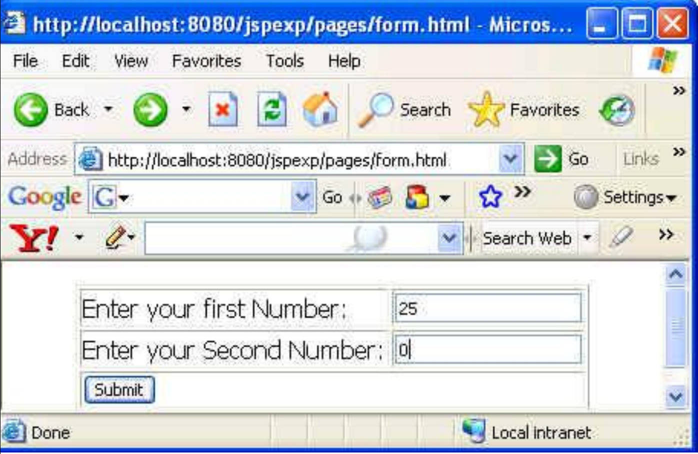
```
<head>
        <style>
        body, p { font-family:Tahoma; font-size:10pt; }
        </style>
</head>
<body>
<%
        int fno;   int sno;
                fno = Integer.parseInt(request.getParameter("fno"));
                sno = Integer.parseInt(request.getParameter("sno"));
                                int div=fno/sno;
%>
<p>Division is : <%= div %></p> <p><a href="form.html">Back</a>.</p>
</body> </html>

<%@ page isErrorPage="true" import="java.io.*" %>
<html>
<head>
        <title>Exceptional Even Occurred!</title>      <style>
        body, p { font-family:Tahoma; font-size:10pt;
    padding-left:30; }
        pre { font-size:8pt; }          </style>
</head>
<body>
<%-- Exception Handler --%> <font color="red">
<%= exception.toString() %><br> </font>
<%
out.println("<!--");
StringWriter sw = new StringWriter();
PrintWriter pw = new PrintWriter(sw);
exception.printStackTrace(pw);
out.print(sw);
sw.close(); pw.close(); out.println("-->");
%>
</body> </html>
```

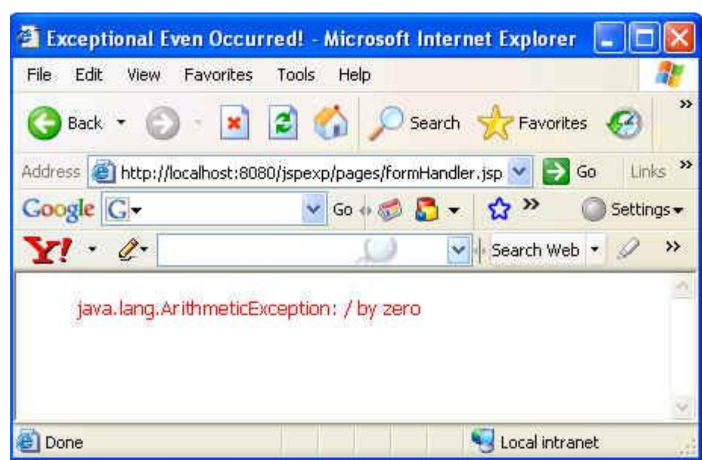
Output of the program:











SESSION MANAGEMENT IN JSP

Http protocol is a stateless protocol, that means that it can't persist the data. Http treats each request as a new request so every time you will send a request you will be considered as a new user. It is not reliable when we are doing any type of transactions or any other related work where persistence of the information is necessary. To remove these obstacles we use session management. In session management whenever a request comes for any resource, a unique token is generated by the server and transmitted to the client by the response object and stored on the client machine as a cookie. We can also say that the process of managing the state of a web based client is through the use of session IDs. Session IDs are used to uniquely identify a client browser, while the server side processes are used to associate the session ID with a level of access. Thus, once a client has successfully authenticated to the web applicatiion, the session ID can be used as a stored authentication voucher so that the client does not have to retype their login information with each page request. Now whenever a request goes from this client again the ID or token will also be passed through the request object so that the server can understand from where the request is coming. Session management can be achieved by using the following thing.

1. **Cookies:** cookies are small bits of textual information that a web server sends to a browser and that browsers returns the cookie when it visits the same site again. In cookie the information is stored in the

form of a name, value pair. By default the cookie is generated. If the user doesn't want to use cookies then it can disable them.

2. **URL rewriting:** In URL rewriting we append some extra information on the end of each URL that identifies the session. This URL rewriting can be used where a cookie is disabled. It is a good practice to use URL rewriting. In this session ID information is embedded in the URL, which is recieved by the application through Http GET requests when the client clicks on the links embedded with a page.

3. **Hidden form fields:** In hidden form fields the html entry will be like this : <input type ="hidden" name = "name" value="">. This means that when you submit the form, the specified name and value will be get included in get or post method. In this session ID information would be embedded within the form as a hidden field and submitted with the Http POST command.

In JSP we have been provided a implicit object session so we don't need to create a object of session explicitly as we do in Servlets. In Jsp the session is by default true. The session is defined inside the directive <%@ page session = "true/false" %>. If we don't declare it inside the jsp page then session will be available to the page, as it is default by true.

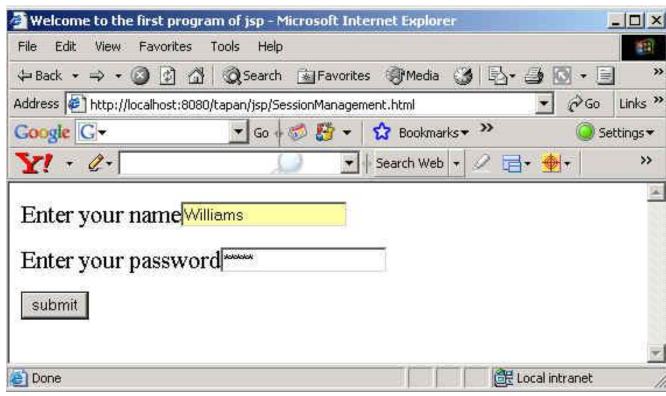
For the convenience to understand the concept of session management we have one program.

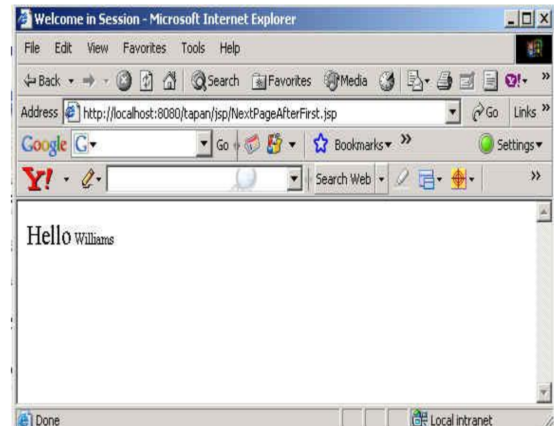
The code of the program is given as:

```
<html>     <head>          <title>Welcome to the first program of jsp</title>     </head>
     <body>
     <form method = "post" action = "FirstPageOfSession.jsp">
 <font size = 6>Enter your name<input type = "text" name = "name"></font><br><br>
 <font size = 6>Enter your password<input type="password" name = "pwd" >
          </font><br><br>
          <input type = "submit" name = "submit" value = "submit" >
     </form>     </body></html>
<%
   String name = request.getParameter("name");
     String password = request.getParameter("pwd");
     if(name.equals("Williams") && password.equals("abcde"))
     {
          session.setAttribute("username",name);
          response.sendRedirect("NextPageAfterFirst.jsp");
     }
     else
     {
          response.sendRedirect("SessionManagement.html");
     }
    %>
<html>
     <head>  <title>Welcome in In the program of URL rewriting</title>     </head>
```

```
<body> <font size = 6>Hello</font> <%= session.getAttribute("username") %>
</body></html>
```

The output of the program is





When the entered values are incorrect, the SessionManagement.html will be displayed again .

<u>MEMORY USAGE CONSIDERATIONS</u>

1. The class takes up at least 8 bytes. So, if usd as **new** Object(); it will allocate 8 bytes on the heap.
2. Each data member takes up 4 bytes, except for long and double which take up 8 bytes. Even if the data member is a byte, it will still take up 4 bytes! In addition, the amount of memory used is increased in 8 byte blocks. So, if you have a class that contains one byte it will take up 8 bytes for the class and 8 bytes for the data, totalling 16 bytes .
3. Arrays are a bit more clever, at least smaller primitives get packed.

UNIT VIII: DATABASE ACCESS

CONTENTS

- Database programming using JDBC
- Javax.sql.*
- Accessing database from jsp page
- Deploying java beans in jsp page
- Introduction to struts framework

DATABASE PROGRAMMING USING JDBC (Java database connectivity)

Database programming has traditionally been a technological Tower Java is supposed to bring us the ability to "write once, compile once, and run anywhere," so it should bring it to us with database programming as well. Java's JDBC API gives us a shared language through which our applications can talk to database engines. Following in the tradition of its other multi-platform APIs such as the AWT, JDBC provides us with a set of interfaces that create a common point at which database applications and database engines can meet. In this chapter, we will discuss the basic interfaces that JDBC provides.

JDBC

JavaSoft developed a single API for database access--JDBC. As part of this process, they kept three main goals in mind:

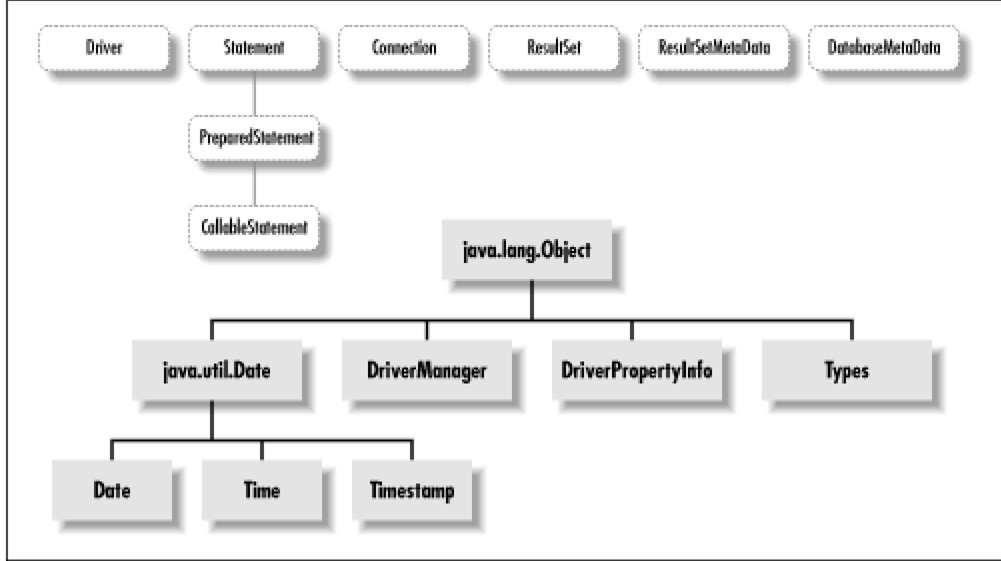
- JDBC should be an SQL-level API.
- JDBC should capitalize on the experience of existing database APIs.
- JDBC should be simple.

An SQL-level API means that JDBC allows us to construct SQL statements and embed them inside Java API calls. In short, you are basically using SQL. But JDBC lets you smoothly translate between the world of the database and the world of the Java application.

JDBC attempts to remain as simple as possible while providing developers with maximum flexibility. A key criterion employed by JavaSoft is simply asking whether database access applications read well. The simple and common tasks use simple interfaces, while more uncommon or bizarre tasks are enabled through extra interfaces. For example, three interfaces handle a vast majority of database access. JDBC nevertheless provides several other interfaces for handling more complex and unusual tasks.

The Structure of JDBC

JDBC accomplishes its goals through a set of Java interfaces, each implemented differently by individual vendors. The set of classes that implement the JDBC interfaces for a particular database engine is called a JDBC driver. In building a database application, you do not have to think about the implementation of these underlying classes at all; the whole point of JDBC is to hide the specifics of each database and let you worry about just your application.



Databases and Drivers

In putting together the examples in this book, I used both an mSQL database If you do not have a corporate pocketbook to back up your database purchase, mSQL is probably the most feasible solution. You should keep in mind, however, that mSQL does not allow you to abort transactions and does not support the stored procedures. Whatever your database choice, you must set up your database engine, create a database, and create the tables shown in the Chapter 3 data model before you can begin writing JDBC code.Once your database engine is installed and your database is all set up, you will need a JDBC driver for that database engine. The more commercial database engines like Oracle have commercial JDBC drivers. Driver categories :

type 1

These drivers use a bridging technology to access a database. The JDBC-ODBC bridge that comes with the JDK 1.1 is a good example of this kind of driver. It provides a gateway to the ODBC API. Implementations of that API in turn do the actual database access. Bridge solutions generally require software to be installed on client systems, meaning that they are not good solutions for applications that do not allow you to install software on the client.

type 2

The type 2 drivers are native API drivers. This means that the driver contains Java code that calls native C or C++ methods provided by the individual database vendors that perform the database access. Again, this solution requires software on the client system.

type 3

Type 3 drivers provide a client with a generic network API that is then translated into database specific access at the server level. In other words, the JDBC driver on the client uses sockets to call a middleware application on the server that translates the client requests into an API specific to the desired driver. As it turns out, this kind of driver is extremely flexible since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

type 4

Using network protocols built into the database engine, type 4 drivers talk directly to the database using Java sockets. This is the most direct pure Java solution. In nearly every case, this type of driver will come only from the database vendor.

Alternatives to JDBC

Without JDBC, only disparate, proprietary database access solutions exist. These proprietary solutions force the developer to build a layer of abstraction on top of them in order to create database independent code. Only after that abstraction layer is complete can the developer move to actually writing the application. In addition, the experience you have with that abstraction layer does not translate immediately to other projects or other employers who are almost certainly using their own abstraction layers to provide access to a variety of database engines.

Connecting to the Database

Any JDBC application we write needs to be able to run from start to finish without ever referencing a specific JDBC implementation. An application uses JDBC as an interface through which it passes all of its databases requests.

The JDBC Support Classes

JDBC provides a handful of other classes and interfaces that support JDBC's core functionality. Many of them are more SQL-friendly extensions of java.util classes like java.sql.Date and java.sql.Numeric. Others are exception classes that get thrown by JDBC calls.

java.sql.Types

The Types class provides constants that identify SQL data types. Each constant that represents an SQL data type that is mapped to an integer is defined by the XOPEN SQL specification. You will see this class used extensively in the next chapter.

java.sql.SQLException

The SQLException class extends the general java.lang.Exception class that provides extra information about a database error. The information provided by a SQLException includes:

- The SQLState string describing the error according to the XOPEN SQLState conventions. The different values of this string are defined in the XOPEN SQL specification.
- The database-specific vendor error code. This code is usually some number that you have to look up in the obscure reference section of your database's documentation. Fortunately, the error should be sufficiently described through the Java Exception class's getMessage() method.
- A chain of exceptions leading up to this one. This is one of the niftier features of this class. Specifically, if you get several errors during the execution of a transaction, you can chain them all together in this class. This is frequently useful when you have exceptions that you want to let the user to know about, but you do not want to stop processing.
  try { Connection connection = DriverManager.getConnection(url, uid, pass); } catch(SQLException e) { e.printStackTrace(); while((e = e.getNextException()) != null) { // while more exceptions e.printStackTrace(); } }

java.sql.SQLWarning and java.sql.DataTruncation

Depending on the driver you are using, non-fatal errors might occur that should not halt application processing. JDBC provides an extension to the SQLException class called SQLWarning. When a JDBC object--like a ResultSet--encounters a warning situation internally, it creates an SQLWarning object and adds it to a list of warnings that it keeps. At any point, you can get the warnings for any JDBC object by repeatedly calling the getWarnings() method until it returns null.

The DataTruncation class is a special kind of warning that a JDBC implementation throws when JDBC unexpectedly truncates a data value. A DataTruncation object is chained as a warning on a read operation and thrown as an exception on a write.

java.sql.Date, java.sql.Time, and java.sql.Timestamp

Portable date handling among database engines can be very complex--each relational database management system (RDBMS) seems to have its own unique way of representing date information. These three classes all extend the functionality of other Java objects to provide a portable representation of their SQL counterparts. The Date and Time classes represent different levels of granularity as well as different means of expressing information already found in the java.util.Date class. The java.sql.Date class, for example, provides methods to express just the date, month, and year, while the Time class works in terms of hours, minutes, and seconds. And finally the Timestamp class takes the java.util.Date class down to nanosecond granularity.

java.sql.DriverPropertyInfo

I can almost guarantee that you will never use this class. It is designed for Rapid Application Development (RAD) tools like Symantec VisualCafe and Borland JBuilder. In order to provide a graphical user interface for rapid prototyping, these tools need to know what properties are required by a given JDBC

implementation in order to connect to the database. Most drivers, for example, need to know the user name and password of the user connecting to the database. That and anything else the driver needs in order to connect to the database will be returned as an array of DriverPropertyInfo objects from the java.sql. Driver getPropertyInfo() method. Development tools can call this method to find out what information they should prompt the user for before connecting to the database.

Database

The database in example consists of a single table of three columns or fields. The database name is "books" and it contains information about **books names & authors**.

Table:books_details

ID	Book Name	Author
1.	Java I/O	Tim Ritchey
2.	Java & XML,2 Edition	Brett McLaughlin
3.	Java Swing, 2nd Edition	Dave Wood, Marc Loy,

Start MYSQL prompt and type this SQL statement & press Enter MYSQL>CREATE DATABASE `books` ;
This will create "books" database.
Now we create table a table "books_details" in database "books".

```
    MYSQL>CREATE TABLE `books_details` ( `id` INT( 11 ) NOT NULL AUTO_INCREMENT ,`book_name`
VARCHAR( 100 ) NOT NULL ,`author` VARCHAR( 100 ) NOT NULL ,  PRIMARY KEY ( `id` ) ) TYPE = MYISAM ;
```

This will create a table "books_details" in database "books"

JSP Code

The following code contains html for user interface & the JSP backend-

```
<%@ page language="java" import="java.sql.*" %>
<%
		String driver = "org.gjt.mm.mysql.Driver";
		Class.forName(driver).newInstance();
		Connection con=null; ResultSet rst=null; Statement stmt=null;
		try{
		String url="jdbc:mysql://localhost/books?user=
<user>&password=<password>";
				con=DriverManager.getConnection(url);
				stmt=con.createStatement();
		}
		catch(Exception e){
				System.out.println(e.getMessage());
		}
		if(request.getParameter("action") != null){
				String bookname=request.getParameter("bookname");
				String author=request.getParameter("author");
```

```
                    stmt.executeUpdate("insert into books_details(book_name,
author) values('"+bookname+"','"+author+"')");
                    rst=stmt.executeQuery("select * from books_details");
                    %>
                    <html>    <body>    <center>
                              <h2>Books List</h2>
<table border="1" cellspacing="0" cellpadding="0">
                              <tr>
                                        <td><b>S.No</b></td>
                                        <td><b>Book Name</b></td>
                                        <td><b>Author</./b></td>
                              </tr>
                                        <%
                                        int no=1;
                                        while(rst.next()){
                                        %>
                                        <tr>
                                          <td><%=no%></td>
                     <td><%=rst.getString("book_name")%></td>
<td> <%=rst.getString("author")
%> </td>
          </tr>
          <%
          no++;
          }
          rst.close();          stmt.close();          con.close();
          %>                              </table>
                              </center>
                    </body>
          </html>
<%}else{%>
          <html>
          <head>
                    <title>Book Entry FormDocument</title>
                    <script language="javascript">
                      function validate(objForm){
                              if(objForm.bookname.value.length==0){
                              alert("Please enter Book Name!");
                              objForm.bookname.focus();
                              return false;
                              }
                              if(objForm.author.value.length==0){
                              alert("Please enter Author name!");
                              objForm.author.focus();
```

```
                                        return false;
                                        }
                                        return true;
                                                    }
                                        </script>
                        </head> <body>   <center>
<form action="BookEntryForm.jsp" method="post"
name="entry" onSubmit="return
 validate(this)">
            <input type="hidden" value="list" name="action">
            <table border="1" cellpadding="0" cellspacing="0">
            <tr>
                        <td>
                                    <table>
                                                <tr>
                                                <td colspan="2" align="center">
<h2>Book Entry Form</h2></td>
                                                </tr>
                                                <tr>
                                                <td colspan="2"> </td>
                                                </tr>
                                                <tr>
                                                <td>Book Name:</td>
                                                <td><input name="bookname" type=
"text" size="50"></td>
                                                </tr>
                                                <tr>
                                                <td>Author:</td><td><input name=
"author" type="text" size="50"></td>
                                                </tr>
                                                <tr>
                                                            <td colspan="2" align="center">
<input type="submit" value="Submit"></td>
                                                            </tr>
                                                </table>
                                    </td>
                        </tr>
            </table>
</form>
                                    </center>
                        </body>
            </html>
<%}%>
```

Declaring Variables: Java is a strongly typed language which means, that variables must be explicitly declared before use and must be declared with the correct data types. In the above example code we declare some variables for making connection. Theses variables are :

```
Connection con=null;
ResultSet rst=null;
Statement stmt=null;
```

The objects of type Connection, ResultSet and Statement are associated with the Java sql. "con" is a Connection type object variable that will hold Connection type object. "rst" is a ResultSet type object variable that will hold a result set returned by a database query. "stmt" is a object variable of Statement .Statement Class methods allow to execute any query.

Connection to database:

```
String driver = "org.gjt.mm.mysql.Driver";Class.forName(driver).newInstance();

String url="jdbc:mysql://localhost/books?user=<userName>&password=<password>";
con=DriverManager.getConnection(url);
```

Executing Query or Accessing data from database:

```
stmt=con.createStatement(); //create a Statement object
rst=stmt.executeQuery("select * from books_details");
```

stmt is the Statement type variable name and **rst** is the RecordSet type variable. A query is always executed on a Statement object.A Statement object is created by calling createStatement() method on connection object **con.** The two most important methods of this Statement interface are executeQuery() and executeUpdate(). The executeQuery() method executes an SQL statement that returns a single ResultSet object. The executeUpdate() method executes an insert, update, and delete SQL statement. The method returns the number of records affected by the SQL statement execution.After creating a Statement ,a method executeQuery() or executeUpdate() is called on Statement object **stmt** and a SQL query string is passed in method executeQuery() or executeUpdate().
This will return a ResultSet **rst** related to the query string.

Reading values from a ResultSet:

```
while(rst.next()){
  %>
<tr><td><%=no%></td><td><%=rst.getString("book_name")%></td><td><%=rst.getString("author")%></
td></tr>  <%
}
```

The ResultSet represents a table-like database result set. A ResultSet object maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row. Therefore, to access the

first row in the ResultSet, you use the next() method. This method moves the cursor to the next record and returns true if the next row is valid, and false if there are no more records in the ResultSet object.

Other important methods are getXXX() methods, where XXX is the data type returned by the method at the specified index, including String, long, and int. The indexing used is 1-based. For example, to obtain the second column of type String, you use the following code:

resultSet.getString(2);

You can also use the getXXX() methods that accept a column name instead of a column index. For instance, the following code retrieves the value of the column LastName of type String.

resultSet.getString("book_name");

The above example shows how you can use the next() method as well as the getString() method. Here you retrieve the 'book_name' and 'author' columns from a table called 'books_details'.

Book Entry Form

Book Name: The Java AWT Reference

Author: John Zukowski

Submit

Books List

S.No	Book Name	Author
1	Java I/O	Tim Ritchey
2	Java & XML,2 Edition	Brett McLaughlin
3	Java Swing, 2nd Edition	Dave Wood, Marc Loy, James Elliott, Brian Cole, Robert Eckstein
4	Java Cookbook, 2nd Edition	Ian F. Darwin
5	Java Web Services Unleashed	Robert J Brunner, Frank Cohen
6	Core Java Data Objects	Sameer Tyagi, Michael Vorburger
7	Java in a Nutshell	David Flanagan
8	Java Web Services in a Nutshell	Kim Topley
9	The Java AWT Reference	John Zukowski

INTRODUCTION TO STRUTS FRAMEWORK

Struts is a framework that promotes the use of the Model-View-Controller architecture for designing large scale applications. The framework includes a set of custom tag libaries and their associated Java classes, along with various utility classes. The most powerful aspect of the Struts framework is its support for creating and processing web-based forms.

Struts Tags

Common Attributes

Almost all tags provided by the Struts framework use the following attributes:

Attribute	Used for
id	the name of a bean for temporary use by the tag
name	the name of a pre-existing bean for use with the tag
property	the property of the bean named in the name attribute for use with the tag
scope	the scope to search for the bean named in the name attribute

Referencing Properties

Bean properties can be referenced in three different ways: simple, nested, or indexed. Shown here are examples for referencing the properties each way:

Reference Method	Example
simple	<!-- uses tutorial.getAnAttribute() --> <bean:write name="tutorial" property="anAttribute"/>
nested	<!-- uses tutorial.getAnAttribute().getAnotherAttribute() --> <bean:write name="tutorial" property="anAttribute.anotherAttribute"/>
indexed	<!-- uses tutorial.getSomeAttributes(3) to access the --> <!-- fourth element of the someAttributes property array --> <bean:write name="tutorial" property="someAttributes[3]"/>
flavorful mix of methods	<!-- uses foo.getSomeAttributes(2).getSomeMoreAttributes(1) --> <bean:write name="foo" property="goo[2].someAttributes[1]"/>

Creating Beans : Beans are created by Java code or tags.

Here is an example of bean creation with Java code:

```
// Creating a Plumber bean in the request scope
Plumber aPlumber = new Plumber();
request.setAttribute("plumber", aPlumber);
```

Beans can be created with the <jsp:useBean></jsp:useBean> tag:

```
<!-- If we want to do <jsp:setProperty ...></jsp:setProperty> or -->
<!-- <jsp:getProperty ... ></jsp:getProperty> -->
<!-- we first need to do a <jsp:useBean ... ></jsp:useBean> -->

<jsp:useBean id="aBean" scope="session" class="java.lang.String">
creating/using a bean in session scope of type java.lang.String
</jsp:useBean>
```

Most useful is the creation of beans with Struts tags:

```
<!-- Constant string bean -->
<bean:define id="greenBean" value="Here is a new constant string bean; pun intended."/>

<!-- Copying an already existent bean, frijole, to a new bean, lima -->
<bean:define id="lima" name="frijole"/>

<!-- Copying an already existent bean, while specifying the class -->
<bean:define id="lima" name="frijole" class="com.SomePackageName.Beans.LimaBean"/>

<!-- Copying a bean property to a different scope -->
<bean:define id="goo" name="foo" property="geeWhiz" scope="request" toScope="application"/>
```

Bean Output

The <bean:message> and <bean:write> tags from the Struts framework will write bean and aplication resources properties into the current HttpResponse object.

<table>
<tr><td><bean:message ... ></td><td>This tag allows locale specific messages to be displayed by looking up the message in the application resources .properties file.

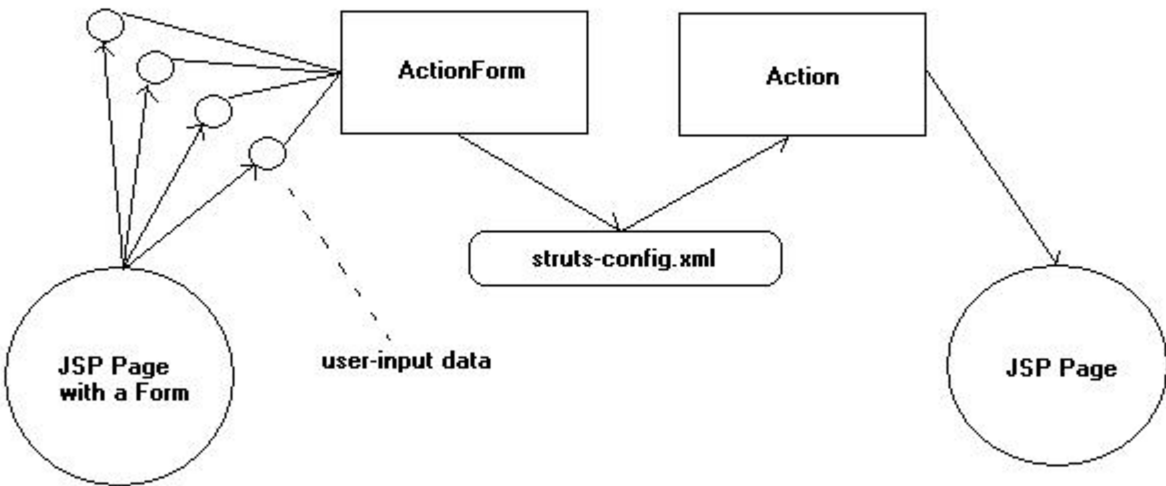
<!-- looks up the error.divisionByZero resource -->
<!-- and writes it to the HttpResponse object -->
<bean:message key="error.divisionByZero"/>

<!-- looks up the prompt.name resource -->
<!-- and writes it to the HttpResponse object; -->
<!-- failing that, it writes the string -->
<!-- contained in the attribute arg0-->
<bean:message key="prompt.name" arg0='Enter a name:'/></td></tr>
<tr><td><bean:write ... ></td><td></td></tr>
</table>

Creating HTML Forms

Quite often information needs to be collected from a user and processed. Without the ability to collect user input, a web application would be useless. In order to get the users information, an html form is used. User input can come from several widgets, such as text fields, text boxes, check boxes, pop-up menus, and radio buttons. The data corresponding to the user input is stored in an ActionForm class. A configuration file called struts-config.xml is used to define exactly how the user input are processed.

The following diagram roughly depicts the use of Struts for using forms.



The Struts html tags are used to generate the widgets in the html that will be used in gathering the users data. There are also tags to create a form element, html body elements, links, images, and other common html elements as well as displaying errors. Below are the tags provided by html section of the Struts framework and a short description of each.