Introduction to JavaScript

JavaScript is a programming language that can be included on web pages to make them more interactive. You can use it to check or modify the contents of forms, change images, open new windows and write dynamic page content. You can even use it with CSS to make DHTML (Dynamic HyperText Markup Language). This allows you to make parts of your web pages appear or disappear or move around on the page. JavaScripts only execute on the page(s) that are on your browser window at any set time. When the user stops viewing that page, any scripts that were running on it are immediately stopped. The only exceptions are cookies or various client side storage APIs, which can be used by many pages to store and pass information between them, even after the pages have been closed.

Before we go any further, let me say; JavaScript has nothing to do with Java. If we are honest, JavaScript, originally nicknamed LiveWire and then LiveScript when it was created by Netscape, should in fact be called ECMAscript as it was renamed when Netscape passed it to the ECMA for standardisation.

JavaScript is a client side, interpreted, object oriented, high level scripting language, while Java is a client side, compiled, object oriented high level language. Now after that mouthful, here's what it means.

Client side

Programs are passed to the computer that the browser is on, and that computer runs them. The alternative is server side, where the program is run on the server and only the results are passed to the computer that the browser is on. Examples of this would be PHP, Perl, ASP, JSP etc.

Interpreted

The program is passed as source code with all the programming language visible. It is then converted into machine code as it is being used. Compiled languages are converted into machine code first then passed around, so you never get to see the original programming language. Java is actually dual half compiled, meaning it is half compiled (to 'byte code') before it is passed, then executed in a virtual machine which converts it to fully compiled code just before use, in order to execute it on the computer's processor. Interpreted languages are generally less fussy about syntax and if you have made mistakes in a part they never use, the mistake usually will not cause you any problems.

Scripting

This is a little harder to define. Scripting languages are often used for performing repetitive tasks. Although they may be complete programming languages, they do not usually go into the depths of complex programs, such as thread and memory management. They may use another program to do the work and simply tell it what to do. They often do not create their own user interfaces, and instead will rely on the other programs to create an interface for them. This is quite accurate for JavaScript. We do not have to tell the browser exactly what to put on the screen for every pixel (though there is a relatively new API known as canvas that makes this possible if needed), we just tell it that we want it to change the document, and it does it. The browser will also take care of

the memory management and thread management, leaving JavaScript free to get on with the things it wants to do.

High level

Written in words that are as close to english as possible. The contrast would be with assembly code, where each command can be directly translated into machine code.

Object oriented

How is JavaScript constructed

The basic part of a script is a variable, literal or object. A variable is a word that represents a piece of text, a number, a boolean true or false value or an object. A literal is the actual number or piece of text or boolean value that the variable represents. An object is a collection of variables held together by a parent variable, or a document component.

The next most important part of a script is an operator. Operators assign literal values to variables or say what type of tests to perform.

The next most important part of a script is a control structure. Control structures say what scripts should be run if a test is satisfied.

Functions collect control structures, actions and assignments together and can be told to run those pieces of script as and when necessary.

The most obvious parts of a script are the actions it performs. Some of these are done with operators but most are done using methods. Methods are a special kind of function and may do things like submitting forms, writing pages or displaying messages.

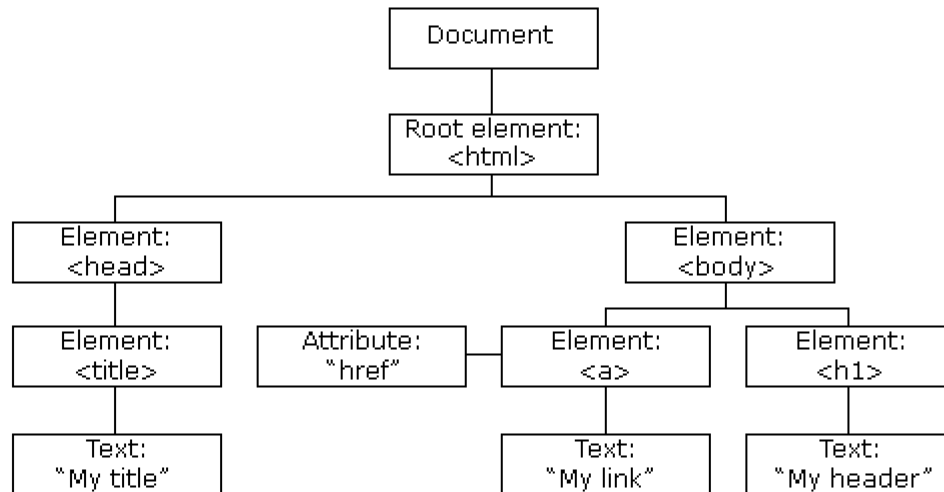
Events can be used to detect actions, usually created by the user, such as moving or clicking the mouse, pressing a key or resetting a form. When triggered, events can be used to run functions.

Lastly and not quite so obvious is referencing. This is about working out what to write to access the contents of objects or even the objects themselves.

As an example, think of the following situation. A person clicks a submit button on a form. When they click the button, we want to check if they have filled out their name in a text box and if they have, we want to submit the form. So, we tell the form to detect the submit event. When the event is triggered, we tell it to run the function that holds together the tests and actions. The function contains a control structure that uses a comparison operator to test the text box to see that it is not empty. Of course we have to work out how to reference the text box first. The text box is an object. One of the variables it holds is the text that is written in the text box. The text written in it is a literal. If the text box is not empty, a method is used that submits the form.

The HTML DOM (Document Object Model)
When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.
The **HTML DOM** model is constructed as a tree of **Objects**:

The HTML DOM Tree of Objects



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

Regular Expression

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the exec and test methods of RegExp, and with the match, replace, search, and split methods of String. This chapter describes JavaScript regular expressions.

Creating a regular expression

You construct a regular expression in one of two ways:

Using a regular expression literal, as follows:

```
var re = /ab+c/;
```

Regular expression literals provide compilation of the regular expression when the script is loaded. When the regular expression will remain constant, use this for better performance.

Or calling the constructor function of the RegExp object, as follows:

```
var re = new RegExp("ab+c");
```

Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

Writing a regular expression pattern

A regular expression pattern is composed of simple characters, such as /abc/, or a combination of simple and special characters, such as /ab*c/ or /Chapter (\d+)\.\d*/. The last example includes parentheses which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in Using parenthesized substring matches.

Using simple patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern /abc/ matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string 'Grab crab' because while it contains the substring 'ab c', it does not contain the exact substring 'abc'.

Using special characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, the pattern includes special characters. For example, the pattern /ab*c/ matches any character combination in which a single 'a' is followed by zero or more 'b's (* means 0 or more occurrences of the preceding item) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

The following table provides a complete list and description of the special characters that can be used in regular expressions.

Special characters in regular expressions.

Character	Meaning
\	Matches according to the following rules: A backslash that precedes a non-special character indicates that the next character is special and is not to be interpreted literally. For example, a 'b' without a preceding '\' generally matches lowercase 'b's wherever they occur. But a '\b' by itself doesn't match any character; it forms the special word boundary character. A backslash that precedes a special character indicates that the next character is not special and should be interpreted literally. For example, the pattern /a*/ relies on the special character '*' to match 0 or more a's. By contrast, the pattern /a*/ removes the specialness of the '*' to enable matches with strings like 'a*'. Do not forget to escape \ itself while using the RegExp("pattern") notation because \ is also an escape character in strings.
^	Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character. For example, /^A/ does not match the 'A' in "an A", but does match the 'A' in "An E". The '^' has a different meaning when it appears as the first character in a character set pattern. See complemented character sets for details and an example.
$	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, /t$/ does not match the 't' in "eater", but does match it in "eat".
*	Matches the preceding character 0 or more times. Equivalent to {0,}.

Special characters in regular expressions.	
Character	**Meaning**
	For example, /bo*/ matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".
+	Matches the preceding character 1 or more times. Equivalent to {1,}. For example, /a+/ matches the 'a' in "candy" and all the a's in "caaaaaaandy", but nothing in "cndy".
?	Matches the preceding character 0 or 1 time. Equivalent to {0,1}. For example, /e?le?/ matches the 'el' in "angel" and the 'le' in "angle" and also the 'l' in "oslo". If used immediately after any of the quantifiers *, +, ?, or {}, makes the quantifier non-greedy (matching the fewest possible characters), as opposed to the default, which is greedy (matching as many characters as possible). For example, applying /\d+/ to "123abc" matches "123". But applying /\d+?/ to that same string matches only the "1". Also used in lookahead assertions, as described in the x(?=y) and x(?!y) entries of this table.
.	(The decimal point) matches any single character except the newline character. For example, /.n/ matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
(x)	Matches 'x' and remembers the match, as the following example shows. The parentheses are called *capturing parentheses*. The '(foo)' and '(bar)' in the pattern /(foo) (bar) \1 \2/ match and remember the first two words in the string "foo bar foo bar". The \1 and \2 in the pattern match the string's last two words. Note that \1, \2, \n are used in the matching part of the regex. In the replacement part of a regex the syntax $1, $2, $n must be used, e.g.: 'bar foo'.replace(/(...) (...)/, '$2 $1').
(?:x)	Matches 'x' but does not remember the match. The parentheses are called *non-capturing parentheses*, and let you define subexpressions for regular expression operators to work with. Consider the sample expression /(?:foo){1,2}/. If the expression was /foo{1,2}/, the {1,2} characters would apply only to the last 'o' in 'foo'. With the non-capturing parentheses, the {1,2} applies to the entire word 'foo'.

Special characters in regular expressions.	
Character	**Meaning**
x(?=y)	Matches 'x' only if 'x' is followed by 'y'. This is called a lookahead. For example, /Jack(?=Sprat)/ matches 'Jack' only if it is followed by 'Sprat'. /Jack(?=Sprat|Frost)/ matches 'Jack' only if it is followed by 'Sprat' or 'Frost'. However, neither 'Sprat' nor 'Frost' is part of the match results.
x(?!y)	Matches 'x' only if 'x' is not followed by 'y'. This is called a negated lookahead. For example, /\d+(?!\.)/ matches a number only if it is not followed by a decimal point. The regular expression /\d+(?!\.)/.exec("3.141") matches '141' but not '3.141'.
x|y	Matches either 'x' or 'y'. For example, /green|red/ matches 'green' in "green apple" and 'red' in "red apple."
{n}	Matches exactly n occurrences of the preceding character. N must be a positive integer. For example, /a{2}/ doesn't match the 'a' in "candy," but it does match all of the a's in "caandy," and the first two a's in "caaandy."
{n,m}	Where n and m are positive integers and n <= m. Matches at least n and at most m occurrences of the preceding character. When m is omitted, it's treated as ∞. For example, /a{1,3}/ matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it.
[xyz]	Character set. This pattern type matches any one of the characters in the brackets, including escape sequences. Special characters like the dot(.) and asterisk (*) are not special inside a character set, so they don't need to be escaped. You can specify a range of characters by using a hyphen, as the following examples illustrate. The pattern [a-d], which performs the same match as [abcd], matches the 'b' in "brisket" and the 'c' in "city". The patterns /[a-z.]+/ and /[\w.]+/ match the entire string "test.i.ng".
[^xyz]	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen. Everything that works in the normal character set also works here. For example, [^abc] is the same as [^a-c]. They initially match 'r' in "brisket" and 'h' in "chop."

Special characters in regular expressions.	
Character	**Meaning**
[\b]	Matches a backspace (U+0008). You need to use square brackets if you want to match a literal backspace character. (Not to be confused with \b.)
\b	Matches a word boundary. A word boundary matches the position where a word character is not followed or preceeded by another word-character. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero. (Not to be confused with [\b].) Examples: /\bm/ matches the 'm' in "moon" ; /oo\b/ does not match the 'oo' in "moon", because 'oo' is followed by 'n' which is a word character; /oon\b/ matches the 'oon' in "moon", because 'oon' is the end of the string, thus not followed by a word character; /\w\b\w/ will never match anything, because a word character can never be followed by both a non-word and a word character. **Note:** JavaScript's regular expression engine defines a specific set of characters to be "word" characters. Any character not in that set is considered a word break. This set of characters is fairly limited: it consists solely of the Roman alphabet in both upper- and lower-case, decimal digits, and the underscore character. Accented characters, such as "é" or "ü" are, unfortunately, treated as word breaks.
\B	Matches a non-word boundary. This matches a position where the previous and next character are of the same type: Either both must be words, or both must be non-words. The beginning and end of a string are considered non-words. For example, /\B../ matches 'oo' in "noonday", and /y\B./ matches 'ye' in "possibly yesterday."
\c*X*	Where *X* is a character ranging from A to Z. Matches a control character in a string. For example, /\cM/ matches control-M (U+000D) in a string.
\d	Matches a digit character. Equivalent to [0-9]. For example, /\d/ or /[0-9]/ matches '2' in "B2 is the suite number."
\D	Matches any non-digit character. Equivalent to [^0-9]. For example, /\D/ or /[^0-9]/ matches 'B' in "B2 is the suite number."
\f	Matches a form feed (U+000C).

Special characters in regular expressions.

Character	Meaning
\n	Matches a line feed (U+000A).
\r	Matches a carriage return (U+000D).
\s	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to [\f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a \u2028\u2029\u202f\u205f\u3000]. For example, /\s\w*/ matches ' bar' in "foo bar."
\S	Matches a single character other than white space. Equivalent to [^ \f\n\r\t\v \u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000]. For example, /\S\w*/ matches 'foo' in "foo bar."
\t	Matches a tab (U+0009).
\v	Matches a vertical tab (U+000B).
\w	Matches any alphanumeric character including the underscore. Equivalent to [A-Za-z0-9_]. For example, /\w/ matches 'a' in "apple," '5' in "$5.28," and '3' in "3D."
\W	Matches any non-word character. Equivalent to [^A-Za-z0-9_]. For example, /\W/ or /[^A-Za-z0-9_]/ matches '%' in "50%."
n	Where *n* is a positive integer, a back reference to the last substring matching the *n* parenthetical in the regular expression (counting left parentheses). For example, /apple(,)\sorange\1/ matches 'apple, orange,' in "apple, orange, cherry, peach."
\0	Matches a NULL (U+0000) character. Do not follow this with another digit, because \0<digits> is an octal escape sequence.
\xhh	Matches the character with the code hh (two hexadecimal digits)
\uhhhh	Matches the character with the code hhhh (four hexadecimal digits).

Escaping user input to be treated as a literal string within a regular expression can be accomplished by simple replacement:

```
function escapeRegExp(string){
  return string.replace(/[.*+?^${}()|[\]\\]/g, "\\$&");
}
```

Using parentheses

Parentheses around any part of the regular expression pattern cause that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use, as described in Using Parenthesized Substring Matches.

For example, the pattern /Chapter (\d+)\.\d*/ illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by one or more numeric characters (\d means any numeric character and + means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with \ means the pattern must look for the literal character '.'), followed by any numeric character 0 or more times (\d means numeric character, * means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapter 3 and 4", because that string does not have a period after the '3'.

To match a substring without causing the matched part to be remembered, within the parentheses preface the pattern with ?:. For example, (?:\d+) matches one or more numeric characters but does not remember the matched characters.

Working with regular expressions

Regular expressions are used with the RegExp methods test and exec and with the String methods match, replace, search, and split. These methods are explained in detail in the JavaScript reference.

Methods that use regular expressions	
Method	**Description**
exec	A RegExp method that executes a search for a match in a string. It returns an array of information.
test	A RegExp method that tests for a match in a string. It returns true or false.
match	A String method that executes a search for a match in a string. It returns an array of

Methods that use regular expressions	
Method	**Description**
	information or null on a mismatch.
search	A String method that tests for a match in a string. It returns the index of the match, or -1 if the search fails.
replace	A String method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
split	A String method that uses a regular expression or a fixed string to break a string into an array of substrings.

When you want to know whether a pattern is found in a string, use the test or search method; for more information (but slower execution) use the exec or match methods. If you use exec or match and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, RegExp. If the match fails, the exec method returns null (which coerces to false).

In the following example, the script uses the exec method to find a match in a string.

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec("cdbbdbsbz");
```

If you do not need to access the properties of the regular expression, an alternative way of creating myArray is with this script:

```
var myArray = /d(b+)d/g.exec("cdbbdbsbz");
```

If you want to construct the regular expression from a string, yet another alternative is this script:

```
var myRe = new RegExp("d(b+)d", "g");
var myArray = myRe.exec("cdbbdbsbz");
```

With these scripts, the match succeeds and returns the array and updates the properties shown in the following table.

<table>
<tr><th>bject</th><th>Property or index</th><th>Description</th><th>In this example</th></tr>
<tr><td rowspan="4">myArray</td><td></td><td>The matched string and all remembered substrings.</td><td>["dbbd", "bb"]</td></tr>
<tr><td>index</td><td>The 0-based index of the match in the input string.</td><td>1</td></tr>
<tr><td>input</td><td>The original string.</td><td>"cdbbdbsbz"</td></tr>
<tr><td>[0]</td><td>The last matched characters.</td><td>"dbbd"</td></tr>
<tr><td rowspan="2">myRe</td><td>lastIndex</td><td>The index at which to start the next match. (This property is set only if the regular expression uses the g option, described in Advanced Searching With Flags.)</td><td>5</td></tr>
<tr><td>source</td><td>The text of the pattern. Updated at the time that the regular expression is created, not executed.</td><td></td></tr>
</table>

As shown in the second form of this example, you can use a regular expression created with an object initializer without assigning it to a variable. If you do, however, every occurrence is a new regular expression. For this reason, if you use this form without assigning it to a variable, you cannot subsequently access the properties of that regular expression. For example, assume you have this script:

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec("cdbbdbsbz");
console.log("The value of lastIndex is " + myRe.lastIndex);

// "The value of lastIndex is 5"
```

However, if you have this script:

```
var myArray = /d(b+)d/g.exec("cdbbdbsbz");
console.log("The value of lastIndex is " + /d(b+)d/g.lastIndex);

// "The value of lastIndex is 0"
```

The occurrences of /d(b+)d/g in the two statements are different regular expression objects and hence have different values for their lastIndex property. If you need to access the properties of a regular expression created with an object initializer, you should first assign it to a variable.

Using parenthesized substring matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, /a(b)c/ matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the Array elements [1], ..., [n].

The number of possible parenthesized substrings is unlimited. The returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

SVCET

The following script uses the replace() method to switch the words in the string. For the replacement text, the script uses the $1 and $2 in the replacement to denote the first and second parenthesized substring matches.

```
var re = /(\w+)\s(\w+)/;
var str = "John Smith";
var newstr = str.replace(re, "$2, $1");
console.log(newstr);
```

This prints "Smith, John".

Advanced searching with flags

Regular expressions have four optional flags that allow for global and case insensitive searching. These flags can be used separately or together in any order, and are included as part of the regular expression.

Regular expression flags	
Flag	**Description**
g	Global search.
i	Case-insensitive search.
m	Multi-line search.
y	Perform a "sticky" search that matches starting at the current position in the target string.

To include a flag with the regular expression, use this syntax:

```
var re = /pattern/flags;
```

or

```
var re = new RegExp("pattern", "flags");
```

Note that the flags are an integral part of a regular expression. They cannot be added or removed later.

For example, re = /\w+\s/g creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

```
var re = /\w+\s/g;
```

```
var str = "fee fi fo fum";
var myArray = str.match(re);
console.log(myArray);
```

This displays ["fee ", "fi ", "fo "]. In this example, you could replace the line:

```
var re = /\w+\s/g;
```

with:

```
var re = new RegExp("\\w+\\s", "g");
```
and get the same result.

The m flag is used to specify that a multiline input string should be treated as multiple lines. If the m flag is used, ^ and $ match at the start or end of any line within the input string instead of the start or end of the entire string.

Date Object

Both the Date(string) constructor and parse() method work on exactly the the same date formats. The difference is that the constructor creates a Date object, while the static Date.parse() method returns a number - more precisely, the number of milliseconds since Jan 1, 1970:?

```
var d1 = new Date("March 1, 2013");
console.log(d1);        //Fri Mar 1 00:00:00 EST 2013
console.log(typeof d1); //object

var d2 = Date.parse("March 1, 2013");
console.log(d2);        //1332302400000
console.log(typeof d2); //number
```

Either of the above will also work for numeric date formats, assuming that you're dealing with a supported format, such as yyyy/MM/dd, yyyy/M/d, yyyy/MM/dd hh:mm, or yyyy/mm/dd hh:mm:ss. Aside from that short list, most other date formats - with the notable exception of long date formats like Mon, January 1, 2000, which make excellent candidates for string parsing - will result in unpredictable results at best. Oddly, according to Wikipedia, the standard Calendar date representation allows both the YYYY-MM-DD and YYYYMMDD formats, as well as the year-month-only YYYY-MM format.

Errors & Exceptions Handling

There are three types of errors in programming: (a) Syntax Errors, (b) Runtime Errors, and (c) Logical Errors.

Syntax Errors

Syntax errors, also called **parsing errors,** occur at compile time in traditional programming languages and at interpret time in JavaScript.

For example, the following line causes a syntax error because it is missing a closing parenthesis.

```
<script type="text/javascript">
  <!--
    window.print(;
  //-->
</script>
```

When a syntax error occurs in JavaScript, only the code contained within the same thread as the syntax error is affected and the rest of the code in other threads gets executed assuming nothing in them depends on the code containing the error.

Runtime Errors

Runtime errors, also called **exceptions,** occur during execution (after compilation/interpretation).

For example, the following line causes a runtime error because here the syntax is correct, but at runtime, it is trying to call a method that does not exist.

```
<script type="text/javascript">
  <!--
    window.printme();
  //-->
</script>
```

Exceptions also affect the thread in which they occur, allowing other JavaScript threads to continue normal execution.

Logical Errors

Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result you expected.

You cannot catch those errors, because it depends on your business requirement what type of logic you want to put in your program.

The try...catch...finally Statement

The latest versions of JavaScript added exception handling capabilities. JavaScript implements the **try...catch...finally** construct as well as the **throw** operator to handle exceptions.

You can **catch** programmer-generated and **runtime** exceptions, but you cannot **catch** JavaScript syntax errors.

Here is the **try...catch...finally** block syntax −

```
<script type="text/javascript">
  <!--
    try {
      // Code to run
      [break;]
    }

    catch ( e ) {
      // Code to run if an exception occurs
      [break;]
    }

    [ finally {
      // Code that is always executed regardless of
      // an exception occurring
    }]
  //-->
</script>
```

The **try** block must be followed by either exactly one **catch** block or one **finally** block (or one of both). When an exception occurs in the **try** block, the exception is placed in **e** and the **catch** block is executed. The optional **finally** block executes unconditionally after try/catch.

Examples

Here is an example where we are trying to call a non-existing function which in turn is raising an exception. Let us see how it behaves without **try...catch**−

```
<html>
  <head>

    <script type="text/javascript">
      <!--
        function myFunc()
        {
          var a = 100;
          alert("Value of variable a is : " + a );
        }
      //-->
    </script>
```

```
  </head>

  <body>
    <p>Click the following to see the result:</p>

    <form>
      <input type="button" value="Click Me" onclick="myFunc();" />
    </form>

  </body>
</html>
```

Event Handler

An event handler executes a segment of a code based on certain events occurring within the application, such as onLoad, onClick. JavaScript event handers can be divided into two parts: interactive event handlers and non-interactive event handlers. An interactive event handler is the one that depends on the user interactivity with the form or the document. For example, onMouseOver is an interactive event handler because it depends on the users action with the mouse. On the other hand non-interactive event handler would be onLoad, because this event handler would automatically execute JavaScript code without the user's interactivity. Here are all the event handlers available in JavaScript:

Event Handler	Used In
onAbort	image
onBlur	select, text, text area
onChange	select, text, textarea
onClick	button, checkbox, radio, link, reset, submit, area
onError	image
onFocus	select, text, testarea
onLoad	windows, image
onMouseOver	link, area
onMouseOut	link, area
onSelect	text, textarea
onSubmit	form
onUnload	window

onAbort:

An onAbort event handler executes JavaScript code when the user aborts loading an image.

```
<HTML>
<TITLE>Example of onAbort Event Handler</TITLE>
<HEAD>
</HEAD>
```

```
<BODY>
<H3>Example of onAbort Event Handler</H3>
<b>Stop the loading of this image and see what happens:</b><p>
<IMG SRC="reaz.gif" onAbort="alert('You stopped the loading the image!')">
</BODY>
</HTML>
```

Here, an alert() method is called using the onAbort event handler when the user aborts loading the image.

onBlur:

An onBlur event handler executes JavaScript code when input focus leaves the field of a text, textarea, or a select option. For windows, frames and framesets the event handler executes JavaScript code when the window loses focus. In windows you need to specify the event handler in the <BODY> attribute. For example:

```
<BODY BGCOLOR='#ffffff' onBlur="document.bgcolor='#000000'">
```

Note: On a Windows platform, the onBlur event does not work with <FRAMESET>.

See Example:

```
<HTML>
<TITLE>Example of onBlur Event Handler</TITLE>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function valid(form){
   var input=0;
   input=document.myform.data.value;
            if (input<0){
                alert("Please input a value that is less than 0");
            }
}
</SCRIPT>
</HEAD>
<BODY>
<H3> Example of onBlur Event Handler</H3>
Try inputting a value less than zero:<br>
<form name="myform">
 <input type="text" name="data" value="" size=10 onBlur="valid(this.form)">
</form>
</BODY>
</HTML>
```

In this example, 'data' is a text field. When a user attempts to leave the field, the onBlur event handler calls the valid() function to confirm that 'data' has a legal value. Note that the keyword *this* is used to refer to the current object.

---

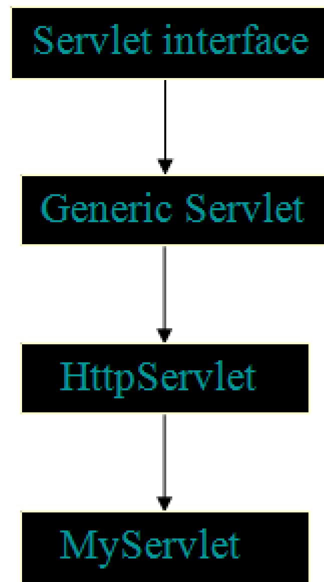
onChange:

The onChange event handler executes JavaScript code when input focus exits the field after the user modifies its text.

See Example:

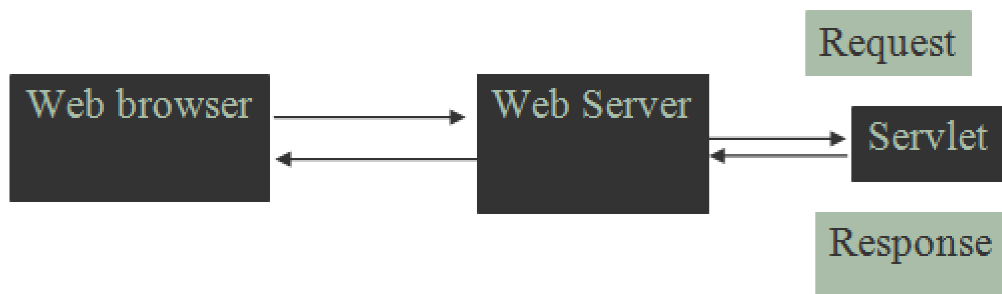
```
<HTML>
<TITLE>Example of onChange Event Handler</TITLE>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function valid(form){
   var input=0;
   input=document.myform.data.value;
   alert("You have changed the value from 10 to " + input );
}
</SCRIPT>
</HEAD>
<BODY>
<H3>Example of onChange Event Handler</H3>
Try changing the value from 10 to something else:<br>
<form name="myform">
 <input type="text" name="data" value="10" size=10 onChange="valid(this.form)">
</form>
</BODY>
</HTML>
```

Servlet is a class, which implements the javax.servlet.Servlet interface. However instead of directly implementing the javax.servlet.Servlet interface we extend a class that has implemented the interface like javax.servlet.GenericServlet or javax.servlet.http.HttpServlet.



Servlet Exceution

This is how a servlet execution takes place when client (browser) makes a request to the webserver.



Servlet architecture includes:

a) **Servlet Interface**
To write a servlet we need to implement Servlet interface. Servlet interface can be implemented directly or indirectly by extending **GenericServlet** or **HttpServlet** class.

b) **Request handling methods**
There are 3 methods defined in Servlet interface: **init(), service() and destroy()**.

The first time a servlet is invoked, the init method is called. It is called only once during the lifetime of a servlet. So, we can put all your initialization code here.

The Service method is used for handling the client request. As the client request reaches to the container it creates a thread of the servlet object, and request and response object are also created. These request and response object are then passed as parameter to the service method, which then process the client request. The service method in turn calls the doGet or doPost methods (if the user has extended the class from HttpServlet).

c) **Number of instances**

Basic Structure of a Servlet

```
public class firstServlet extends HttpServlet {
  public void init() {
    /* Put your initialization code in this method,
     * as this method is called only once */
  }
  public void service() {
    // Service request for Servlet
  }
  public void destroy() {
    // For taking the servlet out of service, this method is called only once
  }
}
```

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet

- The servlet is initialized by calling the **init ()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in details.

The init() method :

The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

```
public void init() throws ServletException {
  // Initialization code...
}
```

The service() method :

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method:

```
public void service(ServletRequest request,
                  ServletResponse response)
      throws ServletException, IOException{
}
```

The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request,
                HttpServletResponse response)
    throws ServletException, IOException {
```

```
    // Servlet code
}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

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

The destroy() method :

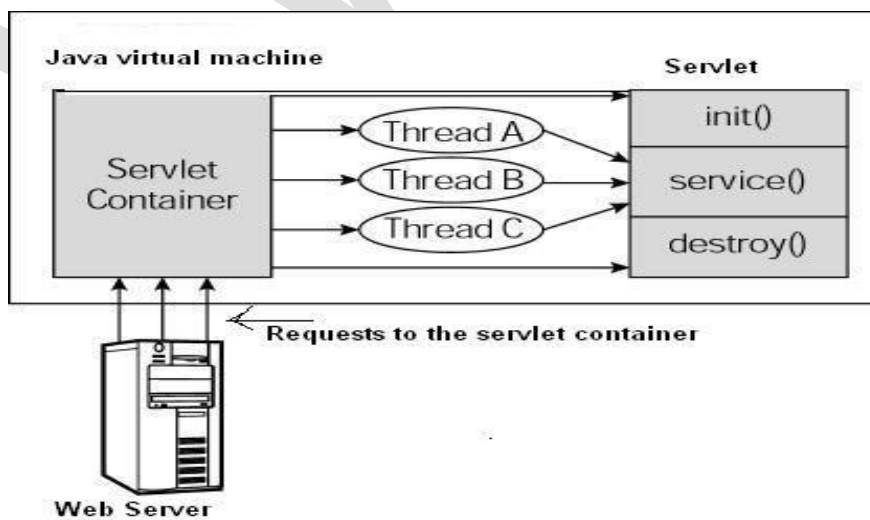
The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
  public void destroy() {
    // Finalization code...
  }
```

Architecture Digram:

The following figure depicts a typical servlet life-cycle scenario.



- First the HTTP requests coming to the server are delegated to the servlet container.
- The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.

session

A session is a conversation between the server and a client. A conversation consists series of continuous request and response.

Why should a session be maintained?

When there is a series of continuous request and response from a same client to a server, the server cannot identify from which client it is getting requests. Because HTTP is a stateless protocol.

When there is a need to maintain the conversational state, session tracking is needed. For example, in a shopping cart application a client keeps on adding items into his cart using multiple requests. When every request is made, the server should identify in which client's cart the item is to be added. So in this scenario, there is a certain need for session tracking.

Solution is, when a client makes a request it should introduce itself by providing unique identifier every time. There are five different methods to achieve this.

Session tracking methods:

1. User authorization
2. Hidden fields
3. URL rewriting
4. Cookies
5. Session tracking API

The first four methods are traditionally used for session tracking in all the server-side technologies. The session tracking API method is provided by the underlying technology (java servlet or PHP or likewise). Session tracking API is built on top of the first four methods.

1. User Authorization

Users can be authorized to use the web application in different ways. Basic concept is that the user will provide username and password to login to the application. Based on that the user can be identified and the session can be maintained.

2. Hidden Fields

<INPUT TYPE=”hidden” NAME=”technology” VALUE=”servlet”>
Hidden fields like the above can be inserted in the webpages and information can be sent to the

server for session tracking. These fields are not visible directly to the user, but can be viewed using view source option from the browsers. This type doesn't need any special configuration from the browser of server and by default available to use for session tracking. This cannot be used for session tracking when the conversation included static resources lik html pages.

3. URL Rewriting

Original URL: http://server:port/servlet/ServletName
Rewritten URL: http://server:port/servlet/ServletName?sessionid=7456
When a request is made, additional parameter is appended with the url. In general added additional parameter will be sessionid or sometimes the userid. It will suffice to track the session. This type of session tracking doesn't need any special support from the browser. Disadvantage is, implementing this type of session tracking is tedious. We need to keep track of the parameter as a chain link until the conversation completes and also should make sure that, the parameter doesn't clash with other application parameters.

4. Cookies

Cookies are the mostly used technology for session tracking. Cookie is a key value pair of information, sent by the server to the browser. This should be saved by the browser in its space in the client computer. Whenever the browser sends a request to that server it sends the cookie along with it. Then the server can identify the client using the cookie.
In java, following is the source code snippet to create a cookie:

```
Cookie cookie = new Cookie("userID", "7456");
res.addCookie(cookie);
```

Session tracking is easy to implement and maintain using the cookies. Disadvantage is that, the users can opt to disable cookies using their browser preferences. In such case, the browser will not save the cookie at client computer and session tracking fails.

5. Session tracking API

Session tracking API is built on top of the first four methods. This is inorder to help the developer to minimize the overhead of session tracking. This type of session tracking is provided by the underlying technology. Lets take the java servlet example. Then, the servlet container manages the session tracking task and the user need not do it explicitly using the java servlets. This is the best of all methods, because all the management and errors related to session tracking will be taken care of by the container itself.

Every client of the server will be mapped with a javax.servlet.http.HttpSession object. Java servlets can use the session object to store and retrieve java objects across the session. Session tracking is at the best when it is implemented using session tracking api.

```
package com.journaldev.servlet.session;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class LoginServlet
 */
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final String userID = "Pankaj";
    private final String password = "journaldev";

    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {

        // get request parameters for userID and password
        String user = request.getParameter("user");
        String pwd = request.getParameter("pwd");

        if(userID.equals(user) && password.equals(pwd)){
            Cookie loginCookie = new Cookie("user",user);
            //setting cookie to expiry in 30 mins
            loginCookie.setMaxAge(30*60);
            response.addCookie(loginCookie);
            response.sendRedirect("LoginSuccess.jsp");
        }else{
            RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");
            PrintWriter out= response.getWriter();
            out.println("<font color=red>Either user name or password is wrong.</font>");
            rd.include(request, response);
        }

    }

}
```

JSP

JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to servlet because it provides more functionality than servlet such as expression language, jstl etc.

A JSP page consists of HTML tags and JSP tags. The jsp pages are easier to maintain than servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tag etc.

Advantage of JSP over Servlet

There are many advantages of JSP over servlet. They are as follows:

1) Extension to Servlet

JSP technology is the extension to servlet technology. We can use all the features of servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.

2) Easy to maintain

JSP can be easily managed because we can easily separate our business logic with presentation logic. In servlet technology, we mix our business logic with the presentation logic.

3) Fast Development: No need to recompile and redeploy

If JSP page is modified, we don't need to recompile and redeploy the project. The servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

4) Less code than Servlet

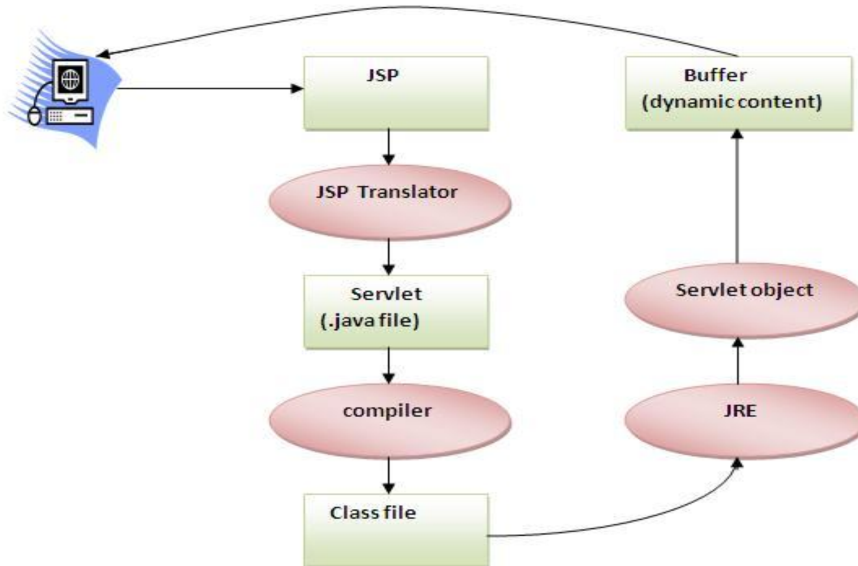
In JSP, we can use a lot of tags such as action tags, jstl, custom tags etc. that reduces the code. Moreover, we can use EL, implicit objects etc.

Life cycle of a JSP Page

The JSP pages follows these phases:

- Translation of JSP Page
- Compilation of JSP Page
- Classloading (class file is loaded by the classloader)
- Instantiation (Object of the Generated Servlet is created).

- Initialization (jspInit() method is invoked by the container).
- Reqeust processing (_jspService() method is invoked by the container).
- Destroy (jspDestroy() method is invoked by the container).



As depicted in the above diagram, JSP page is translated into servlet by the help of JSP translator. The JSP translator is a part of webserver that is responsible to translate the JSP page into servlet. Afterthat Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happens in servlet is performed on JSP later like initialization, committing response to the browser and destroy.

Creating a simple JSP Page

To create the first jsp page, write some html code as given below, and save it by .jsp extension. We have save this file as index.jsp. Put it in a folder and paste the folder in the web-apps directory in apache tomcat to run the jsp page.

index.jsp

Let's see the simple example of JSP, here we are using the scriptlet tag to put java code in the JSP page. We will learn scriptlet tag later.

```
1. <html>
2. <body>
3. <% out.print(2*5); %>
4. </body>
5. </html>
```

It will print **10** on the browser.

How to run a simple JSP Page ?

Follow the following steps to execute this JSP page:

- Start the server
- put the jsp file in a folder and deploy on the server
- visit the browser by the url http://localhost:portno/contextRoot/jspfile e.g. http://localhost:8888/myapplication/index.jsp

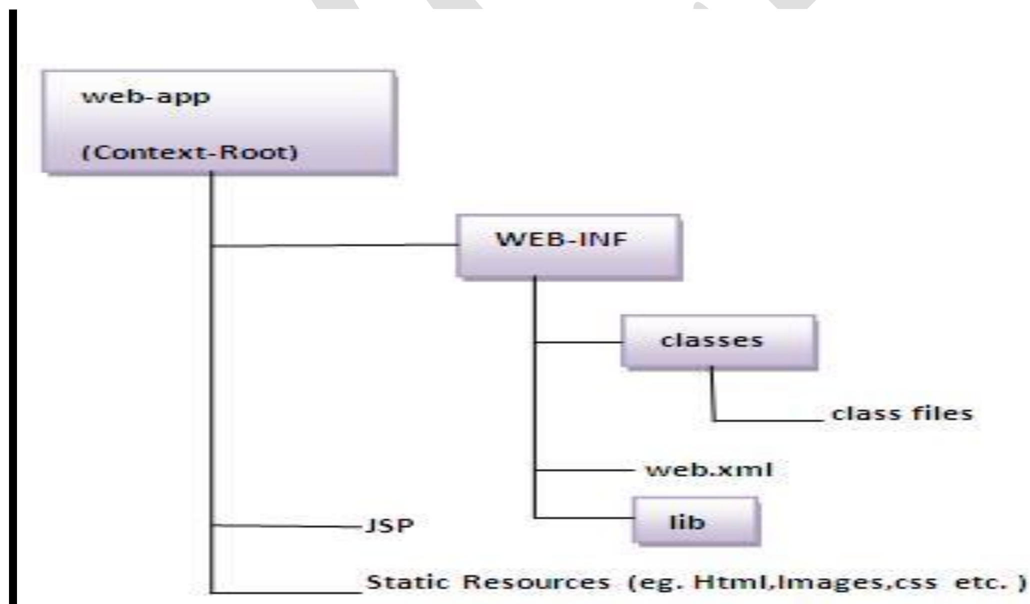
---

Do I need to follow directory structure to run a simple JSP ?

No, there is no need of directory structure if you don't have class files or tld files. For example, put jsp files in a folder directly and deploy that folder.It will be running fine.But if you are using bean class, Servlet or tld file then directory structure is required.

Directory structure of JSP

The directory structure of JSP page is same as servlet. We contains the jsp page outside the WEB-INF folder or in any directory.



Java Server Pages Standard Tag

The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom tags with JSTL tags.

The JSTL tags can be classified, according to their functions, into following JSTL tag library groups that can be used when creating a JSP page:

- **Core Tags**
- **Formatting tags**
- **SQL tags**
- **XML tags**
- **JSTL Functions**

Install JSTL Library:

If you are using Apache Tomcat container then follow the following two simple steps:

- Download the binary distribution from Apache Standard Taglib and unpack the compressed file.
- To use the Standard Taglib from its Jakarta Taglibs distribution, simply copy the JAR files in the distribution's 'lib' directory to your application's webapps\ROOT\WEB-INF\lib directory.

To use any of the libraries, you must include a <taglib> directive at the top of each JSP that uses the library.

Core Tags:

The core group of tags are the most frequently used JSTL tags. Following is the syntax to include JSTL Core library in your JSP:

```
<%@ taglib prefix="c"
        uri="http://java.sun.com/jsp/jstl/core" %>
```

There are following Core JSTL Tags:

Tag	Description
<c:out >	Like <%= ... >, but for expressions.
<c:set >	Sets the result of an expression evaluation in a 'scope'
<c:remove >	Removes a scoped variable (from a particular scope, if specified).

<c:catch>	Catches any Throwable that occurs in its body and optionally exposes it.
<c:if>	Simple conditional tag which evalutes its body if the supplied condition is true.
<c:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<c:when>	Subtag of <choose> that includes its body if its condition evalutes to 'true'.
<c:otherwise >	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'.
<c:import>	Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'.
<c:forEach >	The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality .
<c:forTokens>	Iterates over tokens, separated by the supplied delimeters.
<c:param>	Adds a parameter to a containing 'import' tag's URL.
<c:redirect >	Redirects to a new URL.
<c:url>	Creates a URL with optional query parameters

Formatting tags:

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Web sites. Following is the syntax to include Formatting library in your JSP:

```
<%@ taglib prefix="fmt"
        uri="http://java.sun.com/jsp/jstl/fmt" %>
```

Following is the list of Formatting JSTL Tags:

Tag	Description
<fmt:formatNumber>	To render numerical value with specific precision or format.
<fmt:parseNumber>	Parses the string representation of a number, currency, or percentage.
<fmt:formatDate>	Formats a date and/or time using the supplied styles and pattern
<fmt:parseDate>	Parses the string representation of a date and/or time
<fmt:bundle>	Loads a resource bundle to be used by its tag body.
<fmt:setLocale>	Stores the given locale in the locale configuration variable.
<fmt:setBundle>	Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.

<fmt:timeZone>	Specifies the time zone for any time formatting or parsing actions nested in its body.
<fmt:setTimeZone>	Stores the given time zone in the time zone configuration variable
<fmt:message>	To display an internationalized message.
<fmt:requestEncoding>	Sets the request character encoding

SQL tags:

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as Oracle, mySQL, or Microsoft SQL Server.

Following is the syntax to include JSTL SQL library in your JSP:

```
<%@ taglib prefix="sql"
        uri="http://java.sun.com/jsp/jstl/sql" %>
```

Following is the list of SQL JSTL Tags:

Tag	Description
<sql:setDataSource>	Creates a simple DataSource suitable only for prototyping
<sql:query>	Executes the SQL query defined in its body or through the sql attribute.
<sql:update>	Executes the SQL update defined in its body or through the sql attribute.
<sql:param>	Sets a parameter in an SQL statement to the specified value.
<sql:dateParam>	Sets a parameter in an SQL statement to the specified java.util.Date value.
<sql:transaction >	Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction.

XML tags:

The JSTL XML tags provide a JSP-centric way of creating and manipulating XML documents. Following is the syntax to include JSTL XML library in your JSP.

The JSTL XML tag library has custom tags for interacting with XML data. This includes parsing XML, transforming XML data, and flow control based on XPath expressions.

```
<%@ taglib prefix="x"
        uri="http://java.sun.com/jsp/jstl/xml" %>
```

Before you proceed with the examples, you would need to copy following two XML and XPath related libraries into your <Tomcat Installation Directory>\lib:

- **XercesImpl.jar:** Download it from http://www.apache.org/dist/xerces/j/
- **xalan.jar:** Download it from http://xml.apache.org/xalan-j/index.html

Following is the list of XML JSTL Tags:

Tag	Description
<x:out>	Like <%= ... >, but for XPath expressions.
<x:parse>	Use to parse XML data specified either via an attribute or in the tag body.
<x:set >	Sets a variable to the value of an XPath expression.
<x:if >	Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
<x:forEach>	To loop over nodes in an XML document.
<x:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<x:when >	Subtag of <choose> that includes its body if its expression evalutes to 'true'
<x:otherwise >	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'
<x:transform >	Applies an XSL transformation on a XML document
<x:param >	Use along with the transform tag to set a parameter in the XSLT stylesheet

JSTL Functions:

JSTL includes a number of standard functions, most of which are common string manipulation functions. Following is the syntax to include JSTL Functions library in your JSP:

```
<%@ taglib prefix="fn"
        uri="http://java.sun.com/jsp/jstl/functions" %>
```

Following is the list of JSTL Functions:

Function	Description
fn:contains()	Tests if an input string contains the specified substring.
fn:containsIgnoreCase()	Tests if an input string contains the specified substring in a case insensitive way.
fn:endsWith()	Tests if an input string ends with the specified suffix.
fn:escapeXml()	Escapes characters that could be interpreted as XML markup.
fn:indexOf()	Returns the index withing a string of the first occurrence of a

	specified substring.
fn:join()	Joins all elements of an array into a string.
fn:length()	Returns the number of items in a collection, or the number of characters in a string.
fn:replace()	Returns a string resulting from replacing in an input string all occurrences with a given string.
fn:split()	Splits a string into an array of substrings.
fn:startsWith()	Tests if an input string starts with the specified prefix.
fn:substring()	Returns a subset of a string.
fn:substringAfter()	Returns a subset of a string following a specific substring.
fn:substringBefore()	Returns a subset of a string before a specific substring.
fn:toLowerCase()	Converts all of the characters of a string to lower case.
fn:toUpperCase()	Converts all of the characters of a string to upper case.
fn:trim()	Removes white spaces from both ends of a string.

Creating HTML forms by embedding JSP code

To start off the exploration of HTML forms, it's best to start with a small form and expand from there. Also, it's better to start with a JSP rather than a servlet, because it is easier to write out the HTML. Most of the form handling for JSPs and servlets is identical, so after you know how to retrieve form information from a JSP, you know how to do it from a servlet. Listing 3.1 shows an HTML file containing a simple input form that calls a JSP to handle the form.

```
<html>
<body>

<h1>Please tell me about yourself</h1>

<form action="SimpleFormHandler.jsp" method="get">

Name: <input type="text" name="firstName">
  <input type="text" name="lastName"><br>
Sex:
  <input type="radio" checked name="sex" value="male">Male
  <input type="radio" name="sex" value="female">Female
<p>
What Java primitive type best describes your personality:
<select name="javaType">
  <option value="boolean">boolean</option>
  <option value="byte">byte</option>
  <option value="char" selected>char</option>
```

```
  <option value="double">double</option>
  <option value="float">float</option>
  <option value="int">int</option>
  <option value="long">long</option>
</select>
<br>
<input type="submit">
</form>
</body>
</html>
```

The SimpleFormHandler JSP does little more than retrieve the form variables and print out their values. Listing 3.2 shows the contents of SimpleFormHandler.jsp, which you can see is pretty short.

```
<html>
<body>

<%

// Grab the variables from the form.
  String firstName = request.getParameter("firstName");
  String lastName = request.getParameter("lastName");
  String sex = request.getParameter("sex");
  String javaType = request.getParameter("javaType");
%>
<%-- Print out the variables. --%>
<h1>Hello, <%=firstName%> <%=lastName%>!</h1>
I see that you are <%=sex%>. You know, you remind me of a
<%=javaType%> variable I once knew.

</body>
</html>
```