Unit-I

SCRIPTING.

Web page Designing using HTML, Scripting basics- Client side and server side scripting. Java Script-Object, names, literals, operators and expressions- statements and features- events - windows -documents - frames - data types - built-in functions- Browser object model - Verifying forms.-HTML5-CSS3- HTML 5 canvas - Web site creation using tools.

Web Page Designing using HTML:

Learn HTML Using Notepad or TextEdit

Web pages can be created and modified by using professional HTML editors.

However, for learning HTML we recommend a simple text editor like Notepad (PC) or TextEdit (Mac).

We believe in that using a simple text editor is a good way to learn HTML.

Follow the steps below to create your first web page with Notepad or TextEdit.

Step 1: Open Notepad (PC)

Windows 8 or later:

Open the **Start Screen** (the window symbol at the bottom left on your screen).

Type **Notepad**.

Windows 7 or earlier:

Open **Start** > **Programs** > **Accessories** > **Notepad**

Step 1: Open TextEdit (Mac)

Open **Finder** > **Applications** > **TextEdit**

Also change some preferences to get the application to save files correctly. In **Preferences** > **Format** > choose **"Plain Text"**

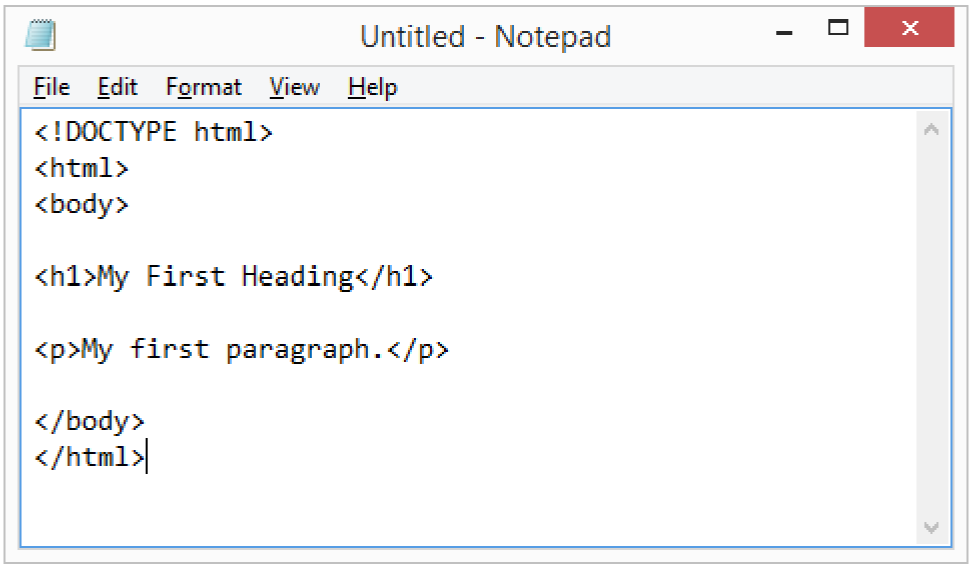
Then under "Open and Save", check the box that says "Display HTML files as HTML code instead of formatted text".

Then open a new document to place the code.

Step 2: Write Some HTML

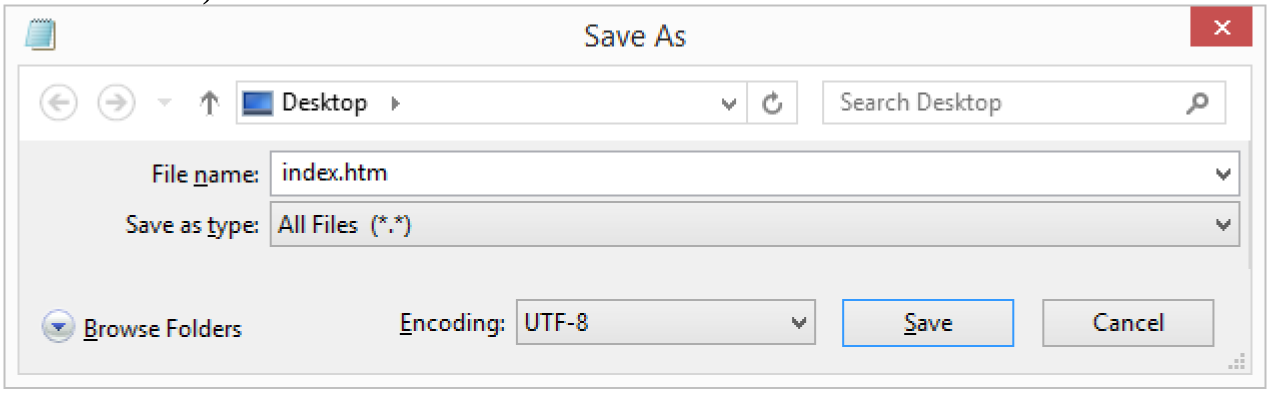
Write or copy the following HTML code into Notepad:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```



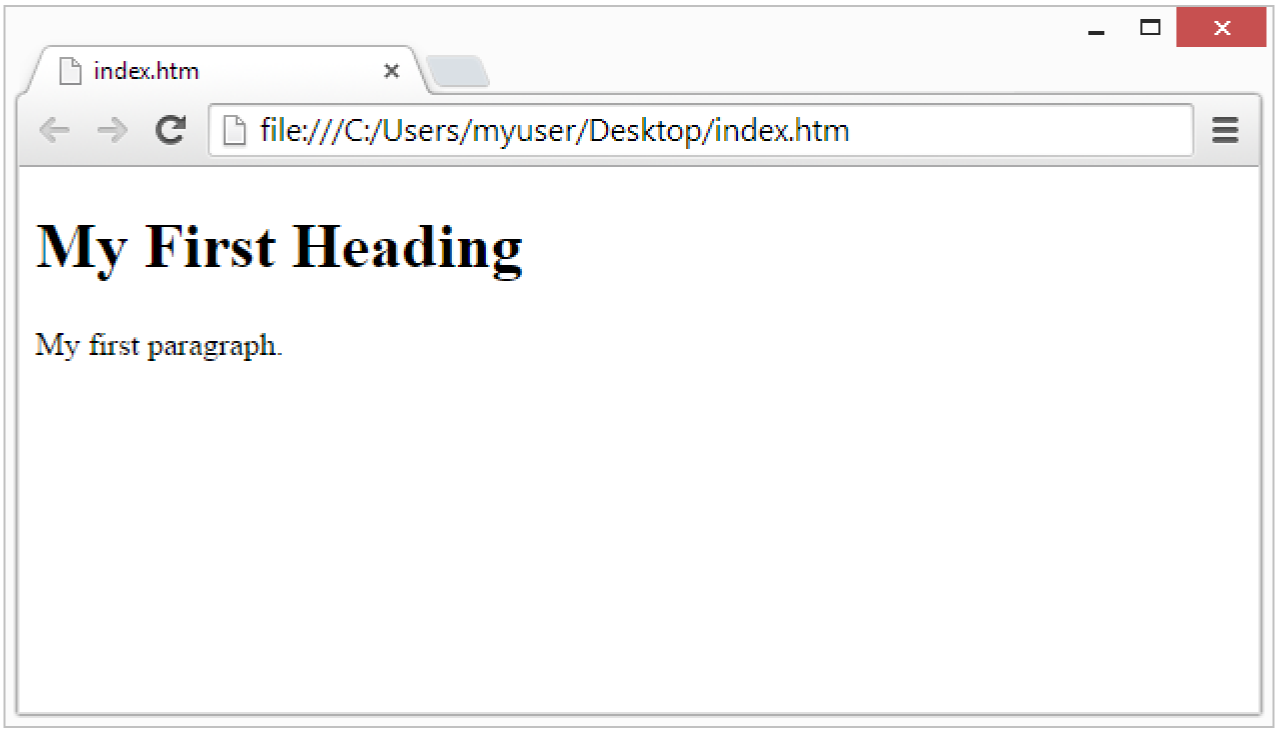
```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

Step 3: Save the HTML Page
Save the file on your computer. Select **File** > **Save as** in the Notepad menu.
Name the file **"index.htm"** and set the encoding to **UTF-8** (which is the preferred encoding for HTML files).



File name: index.htm
Save as type: All Files (*.*)
Encoding: UTF-8

Step 4: View the HTML Page in Your Browser
Open the saved HTML file in your favorite browser (double click on the file, or right-click - and choose "Open with").
The result will look much like this:



My First Heading

My first paragraph.

W3Schools Online Editor - "Try it Yourself"
With our free online editor, you can edit the HTML code and view the result in your browser. It is the perfect tool when you want to **test** code fast. It also has color coding and the ability to save and share code with others:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>This is a Heading</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

Scripting Basics:

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.
Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.
The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.
The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script −

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

Client Side and Server Side Scripting:

1. Client side scripting :
web browsers executes client side scripting. It is use when browsers has all code. Source code used to transfer from web server to user's computer over internet and run directly on browsers. It is also used for validations and functionality for user events.
It allows for more interactivity. It usually performs several actions without going to user. It cannot be basically used to connect to databases on web server. These scripts cannot access file system that resides at web browser. Pages are altered on basis of users choice. It can also used to create "cookies" that store data on user's computer.
2. Server side scripting :
Web servers are used to execute server side scripting. They are basically used to create dynamic pages. It can also access the file system residing at web server. Server-side environment that runs on a scripting language is a web-server.
Scripts can be written in any of a number of server-side scripting language available. It is used to retrieve and generate content for dynamic pages. It is used to require to download plugins. In this load times are generally faster than client-side scripting. When you need to store and retrieve information a database will be used to contain data. It can use huge resources of server. It reduces client-side computation overhead. Server sends pages to request of user/client.

Difference between client side scripting and server side scripting :

CLIENT SIDE SCRIPTING	SERVER SIDE SCRIPTING
Source code is visible to user.	Source code is not visible to user because it's output of server side is a HTML page.

It usually depends on browser and it’s version.	In this any server side technology can be use and it does not depend on client.
It runs on user’s computer.	It runs on web server.
There are many advantages link with this like faster. response times, a more interactive application.	The primary advantage is it’s ability to highly customize, response requirements, access rights based on user.
It does not provide security for data.	It provides more security for data.
It is a technique use in web development in which scripts runs on clients browser.	It is a technique that uses scripts on web server to produce a response that is customized for each clients request.
HTML, CSS and javascript are used.	PHP, Python, Java, Ruby are used.

Java Script Object:

In JavaScript, almost "everything" is an object.

- Booleans can be objects (if defined with the new keyword)
- Numbers can be objects (if defined with the new keyword)
- Strings can be objects (if defined with the new keyword)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are always objects

All JavaScript values, except primitives, are objects.

JavaScript Primitives

A **primitive value** is a value that has no properties or methods.

A **primitive data type** is data that has a primitive value.

JavaScript defines 5 types of primitive data types:

- string
- number
- boolean
- null
- undefined

Primitive values are immutable (they are hardcoded and therefore cannot be changed).

if x = 3.14, you can change the value of x. But you cannot change the value of 3.14.

Objects are Variables

JavaScript variables can contain single values:

Example

```
var person = "John Doe";
```

Objects are variables too. But objects can contain many values.

The values are written as **name : value** pairs (name and value separated by a colon).

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Object Properties

The named values, in JavaScript objects, are called **properties**.

Property	**Value**
firstName	John
lastName	Doe
age	50
eyeColor	Blue

Objects written as name value pairs are similar to:

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

Object Methods

Methods are **actions** that can be performed on objects.

Object properties can be both primitive values, other objects, and functions.

An **object method** is an object property containing a **function definition**.

Property	**Value**
firstName	John
lastName	Doe
age	50
eyeColor	Blue
fullName	function() {return this.firstName + " " + this.lastName;}

You will learn more about methods in the next chapters.

Creating a JavaScript Object

With JavaScript, you can define and create your own objects.

There are different ways to create new objects:

- Define and create a single object, using an object literal.
- Define and create a single object, with the keyword `new`.
- Define an object constructor, and then create objects of the constructed type.

Using an Object Literal

This is the easiest way to create a JavaScript Object.

Using an object literal, you both define and create an object in one statement.

An object literal is a list of name:value pairs (like age:50) inside curly braces {}.
The following example creates a new JavaScript object with four properties:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Spaces and line breaks are not important. An object definition can span multiple lines:

Example

```
var person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

Using the JavaScript Keyword new
The following example also creates a new JavaScript object with four properties:

Example

```
var person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

JavaScript Objects are Mutable
Objects are mutable: They are addressed by reference, not by value.
If person is an object, the following statement will not create a copy of person:

```
var x = person;  // This will not create a copy of person.
```

The object x is **not a copy** of person. It **is** person. Both x and person are the same object.
Any changes to x will also change person, because x and person are the same object.

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}

var x = person;
x.age = 10;           // This will change both x.age and person.age
```

<u>java scripting names, Literals:</u>

JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.
JavaScript was first known as **LiveScript,** but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name **LiveScript**. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers.
The ECMA-262 Specification defined a standard version of the core JavaScript language.

- JavaScript is a lightweight, interpreted programming language.
- Designed for creating network-centric applications.
- Complementary to and integrated with Java.
- Complementary to and integrated with HTML.
- Open and cross-platform

Client-Side JavaScript

Client-side JavaScript is the most common form of the language. The script should be included in or referenced by an HTML document for the code to be interpreted by the browser.

It means that a web page need not be a static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content.

The JavaScript client-side mechanism provides many advantages over traditional CGI server-side scripts. For example, you might use JavaScript to check if the user has entered a valid e-mail address in a form field.

The JavaScript code is executed when the user submits the form, and only if all the entries are valid, they would be submitted to the Web Server.

JavaScript can be used to trap user-initiated events such as button clicks, link navigation, and other actions that the user initiates explicitly or implicitly.

Advantages of JavaScript

The merits of using JavaScript are −

- **Less server interaction** − You can validate user input before sending the page off to the server. This saves server traffic, which means less load on your server.
- **Immediate feedback to the visitors** − They don't have to wait for a page reload to see if they have forgotten to enter something.
- **Increased interactivity** − You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard.
- **Richer interfaces** − You can use JavaScript to include such items as drag-and-drop components and sliders to give a Rich Interface to your site visitors.

Limitations of JavaScript

We cannot treat JavaScript as a full-fledged programming language. It lacks the following important features −

- Client-side JavaScript does not allow the reading or writing of files. This has been kept for security reason.
- JavaScript cannot be used for networking applications because there is no such support available.
- JavaScript doesn't have any multi-threading or multiprocessor capabilities.

Once again, JavaScript is a lightweight, interpreted programming language that allows you to build interactivity into otherwise static HTML pages.

JavaScript Development Tools

One of major strengths of JavaScript is that it does not require expensive development tools. You can start with a simple text editor such as Notepad. Since it is an interpreted language inside the context of a web browser, you don't even need to buy a compiler.

To make our life simpler, various vendors have come up with very nice JavaScript editing tools. Some of them are listed here −

- **Microsoft FrontPage** − Microsoft has developed a popular HTML editor called FrontPage. FrontPage also provides web developers with a number of JavaScript tools to assist in the creation of interactive websites.
- **Macromedia Dreamweaver MX** − Macromedia Dreamweaver MX is a very popular HTML and JavaScript editor in the professional web development crowd. It provides several handy prebuilt JavaScript components, integrates well with databases, and conforms to new standards such as XHTML and XML.
- **Macromedia HomeSite 5** − HomeSite 5 is a well-liked HTML and JavaScript editor from Macromedia that can be used to manage personal websites effectively.

Where is JavaScript Today ?

The ECMAScript Edition 5 standard will be the first update to be released in over four years. JavaScript 2.0 conforms to Edition 5 of the ECMAScript standard, and the difference between the two is extremely minor.
The specification for JavaScript 2.0 can be found on the following site: http://www.ecmascript.org/
Today, Netscape's JavaScript and Microsoft's JScript conform to the ECMAScript standard, although both the languages still support the features that are not a part of the standard.

JavaScript - Syntax

JavaScript can be implemented using JavaScript statements that are placed within the **<script>... </script>** HTML tags in a web page.
You can place the **<script>** tags, containing your JavaScript, anywhere within your web page, but it is normally recommended that you should keep it within the **<head>** tags.
The <script> tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of your JavaScript will appear as follows.

```
<script ...>
  JavaScript code
</script>
```

The script tag takes two important attributes −

- **Language** − This attribute specifies what scripting language you are using. Typically, its value will be javascript. Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.
- **Type** − This attribute is what is now recommended to indicate the scripting language in use and its value should be set to "text/javascript".

So your JavaScript segment will look like −

```
<script language = "javascript" type = "text/javascript">
  JavaScript code
</script>
```

Your First JavaScript Code

Let us take a sample example to print out "Hello World". We added an optional HTML comment that surrounds our JavaScript code. This is to save our code from a browser that does not support JavaScript. The comment ends with a "//-->". Here "//" signifies a comment in JavaScript, so we add that to prevent a browser from reading the end of the HTML comment as a piece of JavaScript code. Next, we call a function **document.write** which writes a string into our HTML document.
This function can be used to write text, HTML, or both. Take a look at the following code.

```
<html>
  <body>
    <script language = "javascript" type = "text/javascript">
      <!--
        document.write("Hello World!")
      //-->
    </script>
  </body>
</html>
```

This code will produce the following result −

```
Hello World!
```

Whitespace and Line Breaks

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

Semicolons are Optional

Simple statements in JavaScript are generally followed by a semicolon character, just as they are in C, C++, and Java. JavaScript, however, allows you to omit this semicolon if each of your statements are placed on a separate line. For example, the following code could be written without semicolons.

```
<script language = "javascript" type = "text/javascript">
  <!--
    var1 = 10
    var2 = 20
  //-->
</script>
```

But when formatted in a single line as follows, you must use semicolons −

```
<script language = "javascript" type = "text/javascript">
  <!--
    var1 = 10; var2 = 20;
  //-->
</script>
```

Note − It is a good programming practice to use semicolons.

Case Sensitivity

JavaScript is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

So the identifiers **Time** and **TIME** will convey different meanings in JavaScript.

NOTE − Care should be taken while writing variable and function names in JavaScript.

Comments in JavaScript

JavaScript supports both C-style and C++-style comments, Thus −

- Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is treated as a comment. This may span multiple lines.
- JavaScript also recognizes the HTML comment opening sequence <!--. JavaScript treats this as a single-line comment, just as it does the // comment.
- The HTML comment closing sequence --> is not recognized by JavaScript so it should be written as //-->.

Example

The following example shows how to use comments in JavaScript.

```
<script language = "javascript" type = "text/javascript">
  <!--
    // This is a comment. It is similar to comments in C++

    /*
    * This is a multi-line comment in JavaScript
    * It is very similar to comments in C Programming
    */
  //-->
</script>
```

Enabling JavaScript in Browsers

All the modern browsers come with built-in support for JavaScript. Frequently, you may need to enable or disable this support manually. This chapter explains the procedure of enabling and disabling JavaScript support in your browsers: Internet Explorer, Firefox, chrome, and Opera.

JavaScript in Internet Explorer

Here are simple steps to turn on or turn off JavaScript in your Internet Explorer −

- Follow **Tools** → **Internet Options** from the menu.
- Select **Security** tab from the dialog box.
- Click the **Custom Level** button.
- Scroll down till you find **Scripting** option.
- Select *Enable* radio button under **Active scripting**.
- Finally click OK and come out

To disable JavaScript support in your Internet Explorer, you need to select **Disable** radio button under **Active scripting**.

JavaScript in Firefox

Here are the steps to turn on or turn off JavaScript in Firefox −

- Open a new tab → type **about: config** in the address bar.
- Then you will find the warning dialog. Select **I'll be careful, I promise!**
- Then you will find the list of **configure options** in the browser.
- In the search bar, type **javascript.enabled**.
- There you will find the option to enable or disable javascript by right-clicking on the value of that option → **select toggle**.

If javascript.enabled is true; it converts to false upon clicking **toogle**. If javascript is disabled; it gets enabled upon clicking toggle.

JavaScript in Chrome

Here are the steps to turn on or turn off JavaScript in Chrome −

- Click the Chrome menu at the top right hand corner of your browser.
- Select **Settings**.
- Click **Show advanced settings** at the end of the page.
- Under the **Privacy** section, click the Content settings button.
- In the "Javascript" section, select "Do not allow any site to run JavaScript" or "Allow all sites to run JavaScript (recommended)".

JavaScript in Opera

Here are the steps to turn on or turn off JavaScript in Opera −

- Follow **Tools** → **Preferences** from the menu.
- Select **Advanced** option from the dialog box.
- Select **Content** from the listed items.
- Select **Enable JavaScript** checkbox.
- Finally click OK and come out.

To disable JavaScript support in your Opera, you should not select the **Enable JavaScript checkbox**.

Warning for Non-JavaScript Browsers

If you have to do something important using JavaScript, then you can display a warning message to the user using **<noscript>** tags.

You can add a **noscript** block immediately after the script block as follows −

```
<html>
  <body>
    <script language = "javascript" type = "text/javascript">
      <!--
        document.write("Hello World!")
      //-->
    </script>
    <noscript>
      Sorry...JavaScript is needed to go ahead.
    </noscript>
  </body>
</html>
```

Now, if the user's browser does not support JavaScript or JavaScript is not enabled, then the message from </noscript> will be displayed on the screen.

JavaScript - Placement in HTML File

There is a flexibility given to include JavaScript code anywhere in an HTML document. However the most preferred ways to include JavaScript in an HTML file are as follows −

- Script in <head>...</head> section.
- Script in <body>...</body> section.
- Script in <body>...</body> and <head>...</head> sections.
- Script in an external file and then include in <head>...</head> section.

In the following section, we will see how we can place JavaScript in an HTML file in different ways.

JavaScript in <head>...</head> section

If you want to have a script run on some event, such as when a user clicks somewhere, then you will place that script in the head as follows −

```
<html>
  <head>
    <script type = "text/javascript">
      <!--
        function sayHello() {
          alert("Hello World")
        }
      //-->
    </script>
  </head>

  <body>
    <input type = "button" onclick = "sayHello()" value = "Say Hello" />
  </body>
</html>
```

This code will produce the following results −

JavaScript in <body>...</body> section

If you need a script to run as the page loads so that the script generates content in the page, then the script goes in the <body> portion of the document. In this case, you would not have any function defined using JavaScript. Take a look at the following code.

```
<html>
  <head>
  </head>

  <body>
    <script type = "text/javascript">
      <!--
        document.write("Hello World")
      //-->
    </script>

    <p>This is web page body </p>
  </body>
</html>
```

This code will produce the following results −

JavaScript in <body> and <head> Sections

You can put your JavaScript code in <head> and <body> section altogether as follows −

```
<html>
  <head>
    <script type = "text/javascript">
      <!--
        function sayHello() {
          alert("Hello World")
        }
      //-->
    </script>
  </head>

  <body>
    <script type = "text/javascript">
      <!--
        document.write("Hello World")
      //-->
    </script>

    <input type = "button" onclick = "sayHello()" value = "Say Hello" />
  </body>
</html>
```

This code will produce the following result −

JavaScript in External File

As you begin to work more extensively with JavaScript, you will be likely to find that there are cases where you are reusing identical JavaScript code on multiple pages of a site.

You are not restricted to be maintaining identical code in multiple HTML files. The **script** tag provides a mechanism to allow you to store JavaScript in an external file and then include it into your HTML files.

Here is an example to show how you can include an external JavaScript file in your HTML code using **script** tag and its **src** attribute.

```
<html>
  <head>
    <script type = "text/javascript" src = "filename.js" ></script>
  </head>

  <body>
    .......
  </body>
</html>
```

To use JavaScript from an external file source, you need to write all your JavaScript source code in a simple text file with the extension ".js" and then include that file as shown above.

For example, you can keep the following content in **filename.js** file and then you can use **sayHello** function in your HTML file after including the filename.js file.

```
function sayHello() {
  alert("Hello World")
}
```

JavaScript - Variables

JavaScript Datatypes

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

JavaScript allows you to work with three primitive data types −

- **Numbers,** eg. 123, 120.50 etc.
- **Strings** of text e.g. "This text string" etc.
- **Boolean** e.g. true or false.

JavaScript also defines two trivial data types, **null** and **undefined,** each of which defines only a single value. In addition to these primitive data types, JavaScript supports a composite data type known as **object**. We will cover objects in detail in a separate chapter.

Note − JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.

JavaScript Variables

Like many other programming languages, JavaScript has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the **var** keyword as follows.

```
<script type = "text/javascript">
  <!--
    var money;
    var name;
  //-->
</script>
```

You can also declare multiple variables with the same **var** keyword as follows −

```
<script type = "text/javascript">
  <!--
    var money, name;
  //-->
</script>
```

Storing a value in a variable is called **variable initialization**. You can do variable initialization at the time of variable creation or at a later point in time when you need that variable.

For instance, you might create a variable named **money** and assign the value 2000.50 to it later. For another variable, you can assign a value at the time of initialization as follows.

```
<script type = "text/javascript">
  <!--
    var name = "Ali";
    var money;
    money = 2000.50;
  //-->
```

```
</script>
```

Note − Use the **var** keyword only for declaration or initialization, once for the life of any variable name in a document. You should not re-declare same variable twice.

JavaScript is **untyped** language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically.

JavaScript Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript variables have only two scopes.

- **Global Variables** − A global variable has global scope which means it can be defined anywhere in your JavaScript code.
- **Local Variables** − A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable. Take a look into the following example.

```
<html>
  <body onload = checkscope();>
    <script type = "text/javascript">
      <!--
        var myVar = "global";      // Declare a global variable
        function checkscope( ) {
          var myVar = "local";    // Declare a local variable
          document.write(myVar);
        }
      //-->
    </script>
  </body>
</html>
```

This produces the following result −

```
local
```

JavaScript Variable Names

While naming your variables in JavaScript, keep the following rules in mind.

- You should not use any of the JavaScript reserved keywords as a variable name. These keywords are mentioned in the next section. For example, **break** or **boolean** variable names are not valid.
- JavaScript variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, **123test** is an invalid variable name but **_123test** is a valid one.
- JavaScript variable names are case-sensitive. For example, **Name** and **name** are two different variables.

JavaScript Reserved Words

A list of all the reserved words in JavaScript are given in the following table. They cannot be used as JavaScript variables, functions, methods, loop labels, or any object names.

abstract	else	instanceof	switch

boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

Operators and Expressions:

What is an Operator?

Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**. JavaScript supports the following types of operators.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators
- Conditional (or ternary) Operators

Lets have a look on all operators one by one.

Arithmetic Operators

JavaScript supports the following arithmetic operators −

Assume variable A holds 10 and variable B holds 20, then −

Sr.No.	Operator & Description
1	**+ (Addition)** Adds two operands **Ex:** A + B will give 30
2	**- (Subtraction)**

	Subtracts the second operand from the first **Ex:** A - B will give -10
3	*** (Multiplication)** Multiply both operands **Ex:** A * B will give 200
4	**/ (Division)** Divide the numerator by the denominator **Ex:** B / A will give 2
5	**% (Modulus)** Outputs the remainder of an integer division **Ex:** B % A will give 0
6	**++ (Increment)** Increases an integer value by one **Ex:** A++ will give 11
7	**-- (Decrement)** Decreases an integer value by one **Ex:** A-- will give 9

Note − Addition operator (+) works for Numeric as well as Strings. e.g. "a" + 10 will give "a10".

Example

The following code shows how to use arithmetic operators in JavaScript.

```
<html>
  <body>

    <script type = "text/javascript">
      <!--
        var a = 33;
        var b = 10;
        var c = "Test";
        var linebreak = "<br />";

        document.write("a + b = ");
        result = a + b;
        document.write(result);
        document.write(linebreak);

        document.write("a - b = ");
        result = a - b;
        document.write(result);
        document.write(linebreak);

        document.write("a / b = ");
        result = a / b;
        document.write(result);
```

```
            document.write(linebreak);

            document.write("a % b = ");
            result = a % b;
            document.write(result);
            document.write(linebreak);

            document.write("a + b + c = ");
            result = a + b + c;
            document.write(result);
            document.write(linebreak);

            a = ++a;
            document.write("++a = ");
            result = ++a;
            document.write(result);
            document.write(linebreak);

            b = --b;
            document.write("--b = ");
            result = --b;
            document.write(result);
            document.write(linebreak);
         //-->
      </script>

      Set the variables to different values and then try...
   </body>
</html>
```

Output

```
a + b = 43
a - b = 23
a / b = 3.3
a % b = 3
a + b + c = 43Test
++a = 35
--b = 8
Set the variables to different values and then try...
```

Comparison Operators

JavaScript supports the following comparison operators −

Assume variable A holds 10 and variable B holds 20, then −

Sr.No.	Operator & Description
1	**= = (Equal)** Checks if the value of two operands are equal or not, if yes, then the condition becomes true. **Ex:** (A == B) is not true.
2	**!= (Not Equal)**

	Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true. **Ex:** (A != B) is true.
3	**> (Greater than)** Checks if the value of the left operand is greater than the value of the right operand, if yes, then the condition becomes true. **Ex:** (A > B) is not true.
4	**< (Less than)** Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true. **Ex:** (A < B) is true.
5	**>= (Greater than or Equal to)** Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A >= B) is not true.
6	**<= (Less than or Equal to)** Checks if the value of the left operand is less than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A <= B) is true.

Example

The following code shows how to use comparison operators in JavaScript.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 10;
        var b = 20;
        var linebreak = "<br />";

        document.write("(a == b) => ");
        result = (a == b);
        document.write(result);
        document.write(linebreak);

        document.write("(a < b) => ");
        result = (a < b);
        document.write(result);
        document.write(linebreak);

        document.write("(a > b) => ");
        result = (a > b);
        document.write(result);
        document.write(linebreak);

        document.write("(a != b) => ");
```

```
            result = (a != b);
            document.write(result);
            document.write(linebreak);

            document.write("(a >= b) => ");
            result = (a >= b);
            document.write(result);
            document.write(linebreak);

            document.write("(a <= b) => ");
            result = (a <= b);
            document.write(result);
            document.write(linebreak);
         //-->
      </script>
      Set the variables to different values and different operators and then try...
   </body>
</html>
```

Output

```
(a == b) => false
(a < b) => true
(a > b) => false
(a != b) => true
(a >= b) => false
a <= b) => true
Set the variables to different values and different operators and then try...
```

Logical Operators

JavaScript supports the following logical operators −

Assume variable A holds 10 and variable B holds 20, then −

Sr.No.	Operator & Description
1	**&& (Logical AND)** If both the operands are non-zero, then the condition becomes true. **Ex:** (A && B) is true.
2	**|| (Logical OR)** If any of the two operands are non-zero, then the condition becomes true. **Ex:** (A || B) is true.
3	**! (Logical NOT)** Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false. **Ex:** ! (A && B) is false.

Example

Try the following code to learn how to implement Logical Operators in JavaScript.

```
<html>
   <body>
      <script type = "text/javascript">
         <!--
         var a = true;
         var b = false;
         var linebreak = "<br />";

         document.write("(a && b) => ");
         result = (a && b);
         document.write(result);
         document.write(linebreak);

         document.write("(a || b) => ");
         result = (a || b);
         document.write(result);
         document.write(linebreak);

         document.write("!(a && b) => ");
         result = (!(a && b));
         document.write(result);
         document.write(linebreak);
       //-->
    </script>
    <p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

```
(a && b) => false
(a || b) => true
!(a && b) => true
Set the variables to different values and different operators and then try...
```

Bitwise Operators

JavaScript supports the following bitwise operators −

Assume variable A holds 2 and variable B holds 3, then −

Sr.No.	Operator & Description
1	**& (Bitwise AND)** It performs a Boolean AND operation on each bit of its integer arguments. **Ex:** (A & B) is 2.
2	**| (BitWise OR)** It performs a Boolean OR operation on each bit of its integer arguments. **Ex:** (A | B) is 3.
3	**^ (Bitwise XOR)** It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. **Ex:** (A ^ B) is 1.
4	**~ (Bitwise Not)** It is a unary operator and operates by reversing all the bits in the operand. **Ex:** (~B) is -4.

5	**<< (Left Shift)** It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on. **Ex:** (A << 1) is 4.
6	**>> (Right Shift)** Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. **Ex:** (A >> 1) is 1.
7	**>>> (Right shift with Zero)** This operator is just like the >> operator, except that the bits shifted in on the left are always zero. **Ex:** (A >>> 1) is 1.

Example

Try the following code to implement Bitwise operator in JavaScript.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 2; // Bit presentation 10
        var b = 3; // Bit presentation 11
        var linebreak = "<br />";

        document.write("(a & b) => ");
        result = (a & b);
        document.write(result);
        document.write(linebreak);

        document.write("(a | b) => ");
        result = (a | b);
        document.write(result);
        document.write(linebreak);

        document.write("(a ^ b) => ");
        result = (a ^ b);
        document.write(result);
        document.write(linebreak);

        document.write("(~b) => ");
        result = (~b);
        document.write(result);
        document.write(linebreak);

        document.write("(a << b) => ");
        result = (a << b);
        document.write(result);
```

```
            document.write(linebreak);

            document.write("(a >> b) => ");
            result = (a >> b);
            document.write(result);
            document.write(linebreak);
         //-->
      </script>
      <p>Set the variables to different values and different operators and then try...</p>
   </body>
</html>
```

```
(a & b) => 2
(a | b) => 3
(a ^ b) => 1
(~b) => -4
(a << b) => 16
(a >> b) => 0
Set the variables to different values and different operators and then try...
```

Assignment Operators

JavaScript supports the following assignment operators −

Sr.No.	Operator & Description
1	**= (Simple Assignment)** Assigns values from the right side operand to the left side operand **Ex:** C = A + B will assign the value of A + B into C
2	**+= (Add and Assignment)** It adds the right operand to the left operand and assigns the result to the left operand. **Ex:** C += A is equivalent to C = C + A
3	**−= (Subtract and Assignment)** It subtracts the right operand from the left operand and assigns the result to the left operand. **Ex:** C -= A is equivalent to C = C - A
4	***= (Multiply and Assignment)** It multiplies the right operand with the left operand and assigns the result to the left operand. **Ex:** C *= A is equivalent to C = C * A
5	**/= (Divide and Assignment)** It divides the left operand with the right operand and assigns the result to the left operand. **Ex:** C /= A is equivalent to C = C / A
6	**%= (Modules and Assignment)** It takes modulus using two operands and assigns the result to the left operand. **Ex:** C %= A is equivalent to C = C % A

Note − Same logic applies to Bitwise operators so they will become like <<=, >>=, >>=, &=, |= and ^=.

Example

Try the following code to implement assignment operator in JavaScript.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 33;
        var b = 10;
        var linebreak = "<br />";

        document.write("Value of a => (a = b) => ");
        result = (a = b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a += b) => ");
        result = (a += b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a -= b) => ");
        result = (a -= b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a *= b) => ");
        result = (a *= b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a /= b) => ");
        result = (a /= b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a %= b) => ");
        result = (a %= b);
        document.write(result);
        document.write(linebreak);
      //-->
    </script>
    <p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

```
Value of a => (a = b) => 10
Value of a => (a += b) => 20
Value of a => (a -= b) => 10
Value of a => (a *= b) => 100
Value of a => (a /= b) => 10
```

Value of a => (a %= b) => 0
Set the variables to different values and different operators and then try...
Miscellaneous Operator
We will discuss two operators here that are quite useful in JavaScript: the **conditional operator** (? :) and the **typeof operator**.

Conditional Operator (? :)

The conditional operator first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation.

Sr.No.	Operator and Description
1	**? : (Conditional)** If Condition is true? Then value X : Otherwise value Y

Example

Try the following code to understand how the Conditional Operator works in JavaScript.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 10;
        var b = 20;
        var linebreak = "<br />";

        document.write ("((a > b) ? 100 : 200) => ");
        result = (a > b) ? 100 : 200;
        document.write(result);
        document.write(linebreak);

        document.write ("((a < b) ? 100 : 200) => ");
        result = (a < b) ? 100 : 200;
        document.write(result);
        document.write(linebreak);
      //-->
    </script>
    <p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

```
((a > b) ? 100 : 200) => 200
((a < b) ? 100 : 200) => 100
Set the variables to different values and different operators and then try...
```

typeof Operator
The **typeof** operator is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.
The *typeof* operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean value and returns true or false based on the evaluation.
Here is a list of the return values for the **typeof** Operator.

Type	String Returned by typeof

Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"
Undefined	"undefined"
Null	"object"

Example

The following code shows how to implement **typeof** operator.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 10;
        var b = "String";
        var linebreak = "<br />";

        result = (typeof b == "string" ? "B is String" : "B is Numeric");
        document.write("Result => ");
        document.write(result);
        document.write(linebreak);

        result = (typeof a == "string" ? "A is String" : "A is Numeric");
        document.write("Result => ");
        document.write(result);
        document.write(linebreak);
      //-->
    </script>
    <p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

```
Result => B is String
Result => A is Numeric
Set the variables to different values and different operators and then try...
```

<u>Statements and Features:</u>

JavaScript Statements

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

Example

```
document.getElementById("demo").innerHTML = "Hello Dolly.";
```

Most JavaScript programs contain many JavaScript statements.
The statements are executed, one by one, in the same order as they are written.
JavaScript programs (and JavaScript statements) are often called JavaScript code.

Semicolons ;
Semicolons separate JavaScript statements.
Add a semicolon at the end of each executable statement:

```
var a, b, c;     // Declare 3 variables
a = 5;           // Assign the value 5 to a
b = 6;           // Assign the value 6 to b
c = a + b;       // Assign the sum of a and b to c
```

When separated by semicolons, multiple statements on one line are allowed:

```
a = 5; b = 6; c = a + b;
```

ADVERTISEMENT

JavaScript White Space
JavaScript ignores multiple spaces. You can add white space to your script to make it more readable.
The following lines are equivalent:

```
var person = "Hege";
var person="Hege";
```

A good practice is to put spaces around operators (= + - * /):

```
var x = y + z;
```

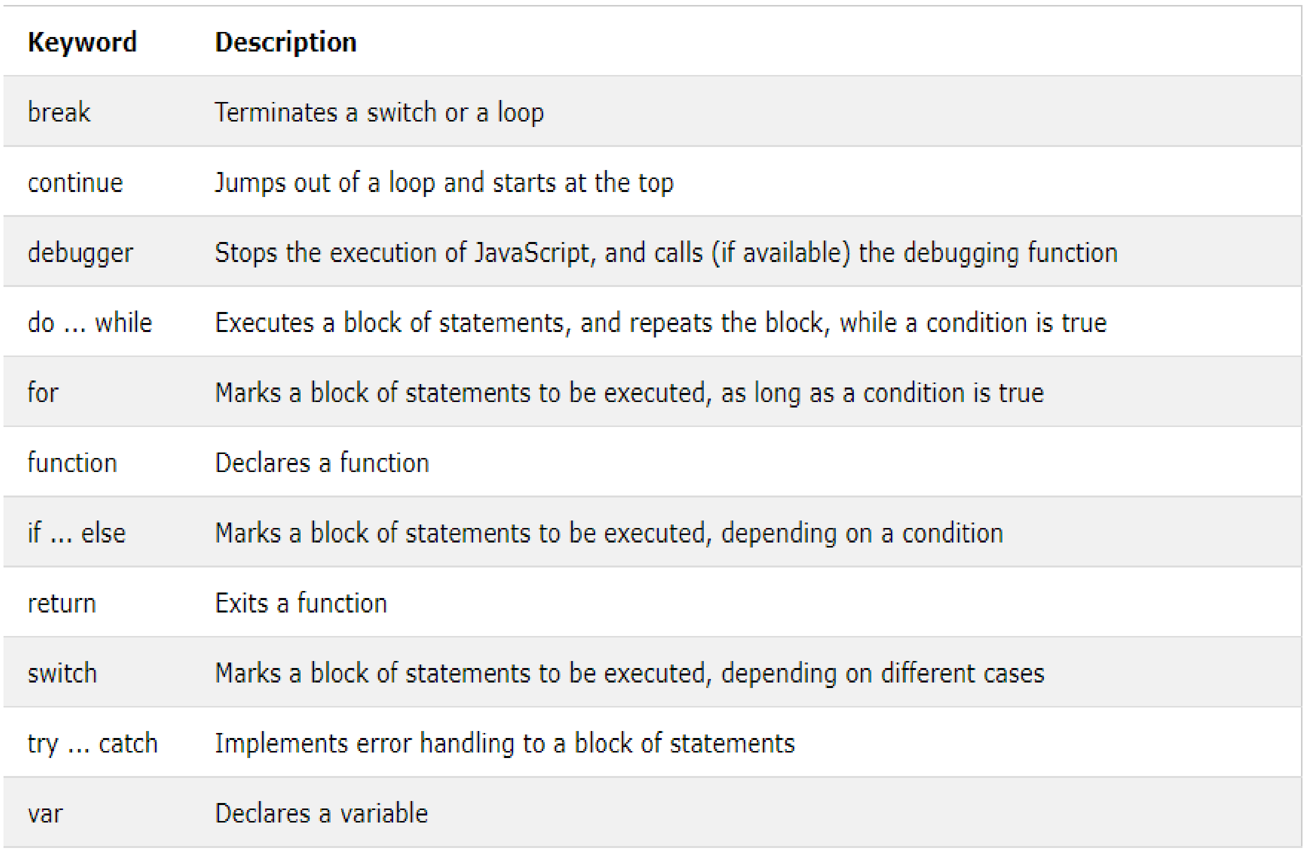
JavaScript Line Length and Line Breaks
For best readability, programmers often like to avoid code lines longer than 80 characters.
If a JavaScript statement does not fit on one line, the best place to break it is after an operator:
Example

```
document.getElementById("demo").innerHTML =
"Hello Dolly!";
```

JavaScript Code Blocks
JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.
The purpose of code blocks is to define statements to be executed together.
One place you will find statements grouped together in blocks, is in JavaScript functions:
Example

```
function myFunction() {
  document.getElementById("demo1").innerHTML = "Hello Dolly!";
  document.getElementById("demo2").innerHTML = "How are you?";
}
```

JavaScript Keywords
JavaScript statements often start with a **keyword** to identify the JavaScript action to be performed.
Visit our Reserved Words reference to view a full list of JavaScript keywords.
Here is a list of some of the keywords you will learn about in this tutorial:



Keyword	Description
break	Terminates a switch or a loop
continue	Jumps out of a loop and starts at the top
debugger	Stops the execution of JavaScript, and calls (if available) the debugging function
do ... while	Executes a block of statements, and repeats the block, while a condition is true
for	Marks a block of statements to be executed, as long as a condition is true
function	Declares a function
if ... else	Marks a block of statements to be executed, depending on a condition
return	Exits a function
switch	Marks a block of statements to be executed, depending on different cases
try ... catch	Implements error handling to a block of statements
var	Declares a variable

JavaScript Events

HTML events are **"things"** that happen to HTML elements.
When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

HTML Events
An HTML event can be something the browser does, or something a user does.
Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.
JavaScript lets you execute code when events are detected.
HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.
With single quotes:

```
<element event='some JavaScript'>
```

With double quotes:

```
<element event="some JavaScript">
```

In the following example, an `onclick` attribute (with code), is added to a `<button>` element:
Example

```
<button onclick="document.getElementById('demo').innerHTML = Date()">The time is?</button>
```

In the example above, the JavaScript code changes the content of the element with id="demo".

In the next example, the code changes the content of its own element
(using **this**.innerHTML):
Example

```
<button onclick="this.innerHTML = Date()">The time is?</button>
```

JavaScript code is often several lines long. It is more common to see event attributes calling functions:
Example

```
<button onclick="displayDate()">The time is?</button>
```

ADVERTISEMENT

Common HTML Events
Here is a list of some common HTML events:
The list is much longer: W3Schools JavaScript Reference HTML DOM Events.

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

What can JavaScript Do?
Event handlers can be used to handle, and verify, user input, user actions, and browser actions:

- Things that should be done every time a page loads
- Things that should be done when the page is closed
- Action that should be performed when a user clicks a button
- Content that should be verified when a user inputs data
- And more ...

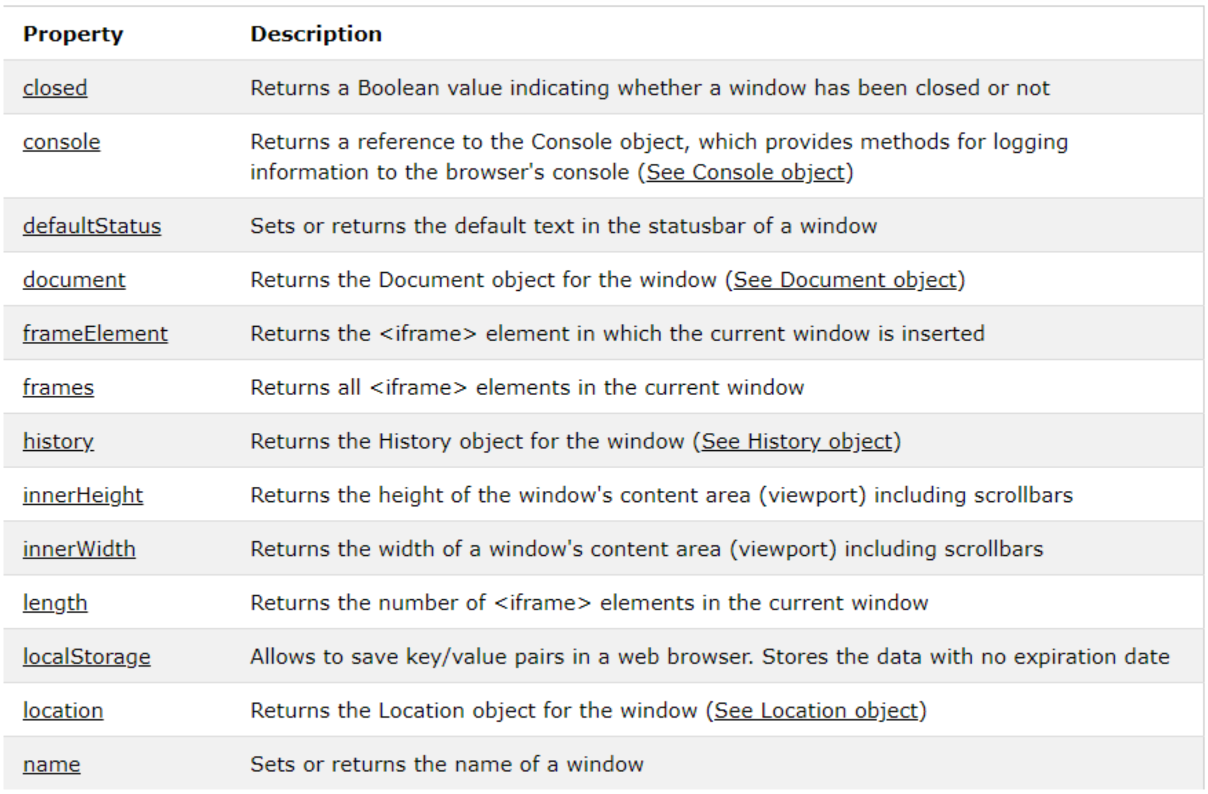
Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly
- HTML event attributes can call JavaScript functions
- You can assign your own event handler functions to HTML elements
- You can prevent events from being sent or being handled
- And more ...

Windows:

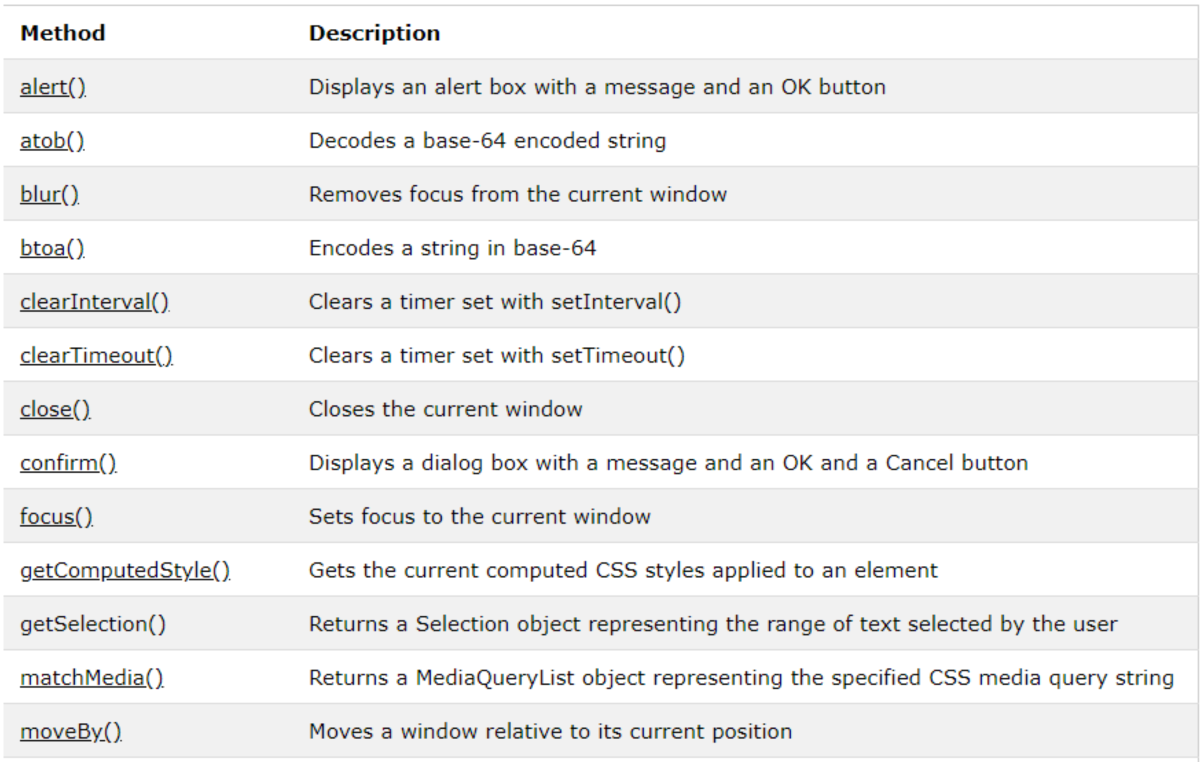
The Window Object
Window Object
The window object represents an open window in a browser.
If a document contain frames (<iframe> tags), the browser creates one window object for the HTML document, and one additional window object for each frame.
Note: There is no public standard that applies to the Window object, but all major browsers support it.

Window Object Properties



Property	Description
closed	Returns a Boolean value indicating whether a window has been closed or not
console	Returns a reference to the Console object, which provides methods for logging information to the browser's console (See Console object)
defaultStatus	Sets or returns the default text in the statusbar of a window
document	Returns the Document object for the window (See Document object)
frameElement	Returns the <iframe> element in which the current window is inserted
frames	Returns all <iframe> elements in the current window
history	Returns the History object for the window (See History object)
innerHeight	Returns the height of the window's content area (viewport) including scrollbars
innerWidth	Returns the width of a window's content area (viewport) including scrollbars
length	Returns the number of <iframe> elements in the current window
localStorage	Allows to save key/value pairs in a web browser. Stores the data with no expiration date
location	Returns the Location object for the window (See Location object)
name	Sets or returns the name of a window

Window Object Methods



Method	Description
alert()	Displays an alert box with a message and an OK button
atob()	Decodes a base-64 encoded string
blur()	Removes focus from the current window
btoa()	Encodes a string in base-64
clearInterval()	Clears a timer set with setInterval()
clearTimeout()	Clears a timer set with setTimeout()
close()	Closes the current window
confirm()	Displays a dialog box with a message and an OK and a Cancel button
focus()	Sets focus to the current window
getComputedStyle()	Gets the current computed CSS styles applied to an element
getSelection()	Returns a Selection object representing the range of text selected by the user
matchMedia()	Returns a MediaQueryList object representing the specified CSS media query string
moveBy()	Moves a window relative to its current position

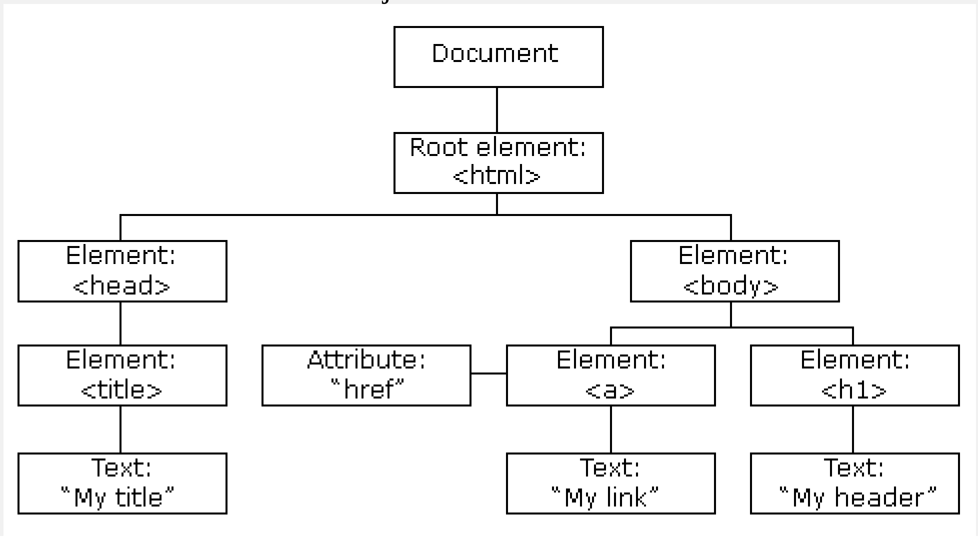
Documents:

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

What You Will Learn

In the next chapters of this tutorial you will learn:

- How to change the content of HTML elements
- How to change the style (CSS) of HTML elements
- How to react to HTML DOM events
- How to add and delete HTML elements

What is the DOM?

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

What is the HTML DOM?

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

Frames:

Window frames Property

Example

Change the location of the first <iframe> element (index 0) in the current window:

```
window.frames[0].location = "https://www.w3schools.com/jsref/";
```

More "Try it Yourself" examples below.

Definition and Usage

The frames property returns an array-like object, which represents all <iframe> elements in the current window.

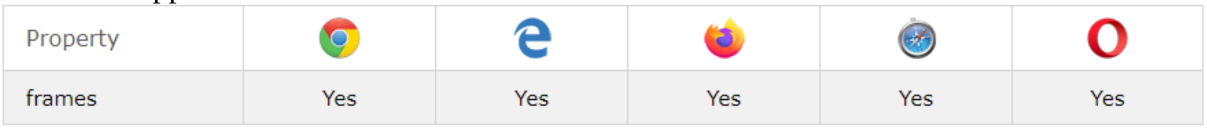
The <iframe> elements can be accessed by index numbers. The index starts at 0.

Tip: Use frames.length to find the number of frames.

Note: This property will also work for <frame> elements. However, the <frame> element is not supported in HTML5.

This property is read-only.

Browser Support



Property					
frames	Yes	Yes	Yes	Yes	Yes

Syntax

```
window.frames
```

More Examples

Example

Loop through all frames on a page, and change the location of all frames to "w3schools.com":

```
var frames = window.frames;
var i;

for (i = 0; i < frames.length; i++) {
  frames[i].location = "https://www.w3schools.com";
}
```

Datatypes:

JavaScript Data Types

JavaScript variables can hold many **data types**: numbers, strings, objects and more:

```
var length = 16;                               // Number
var lastName = "Johnson";                      // String
var x = {firstName:"John", lastName:"Doe"};    // Object
```

The Concept of Data Types

In programming, data types is an important concept.

To be able to operate on variables, it is important to know something about the type.

Without data types, a computer cannot safely solve this:

```
var x = 16 + "Volvo";
```

Does it make any sense to add "Volvo" to sixteen? Will it produce an error or will it produce a result?

JavaScript will treat the example above as:

```
var x = "16" + "Volvo";
```
When adding a number and a string, JavaScript will treat the number as a string.

Example
```
var x = 16 + "Volvo";
```
Example
```
var x = "Volvo" + 16;
```
JavaScript evaluates expressions from left to right. Different sequences can produce different results:

JavaScript:
```
var x = 16 + 4 + "Volvo";
```
Result:
```
20Volvo
```
JavaScript:
```
var x = "Volvo" + 16 + 4;
```
Result:
```
Volvo164
```
In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo".
In the second example, since the first operand is a string, all operands are treated as strings.

JavaScript Types are Dynamic

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

Example
```
var x;           // Now x is undefined
x = 5;           // Now x is a Number
x = "John";      // Now x is a String
```

JavaScript Strings

A string (or a text string) is a series of characters like "John Doe".
Strings are written with quotes. You can use single or double quotes:

Example
```
var carName1 = "Volvo XC60";   // Using double quotes
var carName2 = 'Volvo XC60';   // Using single quotes
```
You can use quotes inside a string, as long as they don't match the quotes surrounding the

Example
```
var answer1 = "It's alright";             // Single quote inside double quotes
var answer2 = "He is called 'Johnny'";    // Single quotes inside double quotes
var answer3 = 'He is called "Johnny"';    // Double quotes inside single quotes
```
You will learn more about strings later in this tutorial.

JavaScript Numbers

JavaScript has only one type of numbers.
Numbers can be written with, or without decimals:

Example
```
var x1 = 34.00;     // Written with decimals
var x2 = 34;        // Written without decimals
```
Extra large or extra small numbers can be written with scientific (exponential) notation:

Example
```
var y = 123e5;      // 12300000
var z = 123e-5;     // 0.00123
```
You will learn more about numbers later in this tutorial.

JavaScript Booleans

Booleans can only have two values: true or false.

Example

```
var x = 5;
var y = 5;
var z = 6;
(x == y)       // Returns true
(x == z)       // Returns false
```

Booleans are often used in conditional testing.
You will learn more about conditional testing later in this tutorial.

JavaScript Arrays

JavaScript arrays are written with square brackets.
Array items are separated by commas.
The following code declares (creates) an array called cars, containing three items (car names):

Example

```
var cars = ["Saab", "Volvo", "BMW"];
```

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.
You will learn more about arrays later in this tutorial.

JavaScript Objects

JavaScript objects are written with curly braces {}.
Object properties are written as name:value pairs, separated by commas.

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.
You will learn more about objects later in this tutorial.

The typeof Operator

You can use the JavaScript typeof operator to find the type of a JavaScript variable.
The typeof operator returns the type of a variable or an expression:

Example

```
typeof ""             // Returns "string"
typeof "John"         // Returns "string"
typeof "John Doe"     // Returns "string"
```

Example

```
typeof 0              // Returns "number"
typeof 314            // Returns "number"
typeof 3.14           // Returns "number"
typeof (3)            // Returns "number"
typeof (3 + 4)        // Returns "number"
```

Undefined

In JavaScript, a variable without a value, has the value undefined. The type is also undefined.

Example

```
var car;    // Value is undefined, type is undefined
```

Any variable can be emptied, by setting the value to undefined. The type will also be undefined.

Example

```
car = undefined;    // Value is undefined, type is undefined
```

Empty Values

An empty value has nothing to do with undefined.

An empty string has both a legal value and a type.

Example

```
var car = "";    // The value is "", the typeof is "string"
```

Null

In JavaScript null is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of null is an object.

You can empty an object by setting it to null:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;    // Now value is null, but type is still an object
```

You can also empty an object by setting it to undefined:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = undefined;   // Now both value and type is undefined
```

Difference Between Undefined and Null

undefined and null are equal in value but different in type:

```
typeof undefined          // undefined
typeof null               // object

null === undefined        // false
null == undefined         // true
```

Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The typeof operator can return one of these primitive types:

- string
- number
- boolean
- undefined

Example

```
typeof "John"             // Returns "string"
typeof 3.14               // Returns "number"
typeof true               // Returns "boolean"
typeof false              // Returns "boolean"
typeof x                  // Returns "undefined" (if x has no value)
```

Complex Data

The typeof operator can return one of two complex types:

- function

- object

The typeof operator returns "object" for objects, arrays, and null.
The typeof operator does not return "object" for functions.

Example

```
typeof {name:'John', age:34} // Returns "object"
typeof [1,2,3,4]             // Returns "object" (not "array", see note below)
typeof null                  // Returns "object"
typeof function myFunc(){}   // Returns "function"
```

<u>Built-in Functions:</u>

Number Methods

The Number object contains only the default methods that are part of every object's definition.

Sr.No.	Method & Description
1	constructor() Returns the function that created this object's instance. By default this is the Number object.
2	toExponential() Forces a number to display in exponential notation, even if the number is in the range in which JavaScript normally uses standard notation.
3	toFixed() Formats a number with a specific number of digits to the right of the decimal.
4	toLocaleString() Returns a string value version of the current number in a format that may vary according to a browser's locale settings.
5	toPrecision() Defines how many total digits (including digits to the left and right of the decimal) to display of a number.
6	toString() Returns the string representation of the number's value.
7	valueOf() Returns the number's value.

Boolean Methods

Here is a list of each method and its description.

Sr.No.	Method & Description
1	toSource() Returns a string containing the source of the Boolean object; you can use this string to create an equivalent object.
2	toString()

	Returns a string of either "true" or "false" depending upon the value of the object.
3	valueOf() Returns the primitive value of the Boolean object.

String Methods
Here is a list of each method and its description.

Sr.No.	Method & Description
1	charAt() Returns the character at the specified index.
2	charCodeAt() Returns a number indicating the Unicode value of the character at the given index.
3	concat() Combines the text of two strings and returns a new string.
4	indexOf() Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
5	lastIndexOf() Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
6	localeCompare() Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.
7	length() Returns the length of the string.
8	match() Used to match a regular expression against a string.
9	replace() Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
10	search() Executes the search for a match between a regular expression and a specified string.
11	slice() Extracts a section of a string and returns a new string.
12	split() Splits a String object into an array of strings by separating the string into substrings.
13	substr()

	Returns the characters in a string beginning at the specified location through the specified number of characters.
14	substring() Returns the characters in a string between two indexes into the string.
15	toLocaleLowerCase() The characters within a string are converted to lower case while respecting the current locale.
16	toLocaleUpperCase() The characters within a string are converted to upper case while respecting the current locale.
17	toLowerCase() Returns the calling string value converted to lower case.
18	toString() Returns a string representing the specified object.
19	toUpperCase() Returns the calling string value converted to uppercase.
20	valueOf() Returns the primitive value of the specified object.

String HTML wrappers

Here is a list of each method which returns a copy of the string wrapped inside the appropriate HTML tag.

Sr.No.	**Method & Description**
1	anchor() Creates an HTML anchor that is used as a hypertext target.
2	big() Creates a string to be displayed in a big font as if it were in a <big> tag.
3	blink() Creates a string to blink as if it were in a <blink> tag.
4	bold() Creates a string to be displayed as bold as if it were in a <b> tag.
5	fixed() Causes a string to be displayed in fixed-pitch font as if it were in a <tt> tag
6	fontcolor() Causes a string to be displayed in the specified color as if it were in a <font color="color"> tag.

7	fontsize() Causes a string to be displayed in the specified font size as if it were in a <font size="size"> tag.
8	italics() Causes a string to be italic, as if it were in an <i> tag.
9	link() Creates an HTML hypertext link that requests another URL.
10	small() Causes a string to be displayed in a small font, as if it were in a <small> tag.
11	strike() Causes a string to be displayed as struck-out text, as if it were in a <strike> tag.
12	sub() Causes a string to be displayed as a subscript, as if it were in a <sub> tag
13	sup() Causes a string to be displayed as a superscript, as if it were in a <sup> tag

Array Methods

Here is a list of each method and its description.

Sr.No.	**Method & Description**
1	concat() Returns a new array comprised of this array joined with other array(s) and/or value(s).
2	every() Returns true if every element in this array satisfies the provided testing function.
3	filter() Creates a new array with all of the elements of this array for which the provided filtering function returns true.
4	forEach() Calls a function for each element in the array.
5	indexOf() Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.
6	join() Joins all elements of an array into a string.
7	lastIndexOf() Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.

8	map() Creates a new array with the results of calling a provided function on every element in this array.
9	pop() Removes the last element from an array and returns that element.
10	push() Adds one or more elements to the end of an array and returns the new length of the array.
11	reduce() Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
12	reduceRight() Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.
13	reverse() Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.
14	shift() Removes the first element from an array and returns that element.
15	slice() Extracts a section of an array and returns a new array.
16	some() Returns true if at least one element in this array satisfies the provided testing function.
17	toSource() Represents the source code of an object
18	sort() Sorts the elements of an array.
19	splice() Adds and/or removes elements from an array.
20	toString() Returns a string representing the array and its elements.
21	unshift() Adds one or more elements to the front of an array and returns the new length of the array.

Date Methods

Here is a list of each method and its description.

Sr.No.	**Method & Description**

1	Date() Returns today's date and time
2	getDate() Returns the day of the month for the specified date according to local time.
3	getDay() Returns the day of the week for the specified date according to local time.
4	getFullYear() Returns the year of the specified date according to local time.
5	getHours() Returns the hour in the specified date according to local time.
6	getMilliseconds() Returns the milliseconds in the specified date according to local time.
7	getMinutes() Returns the minutes in the specified date according to local time.
8	getMonth() Returns the month in the specified date according to local time.
9	getSeconds() Returns the seconds in the specified date according to local time.
10	getTime() Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC.
11	getTimezoneOffset() Returns the time-zone offset in minutes for the current locale.
12	getUTCDate() Returns the day (date) of the month in the specified date according to universal time.
13	getUTCDay() Returns the day of the week in the specified date according to universal time.
14	getUTCFullYear() Returns the year in the specified date according to universal time.
15	getUTCHours() Returns the hours in the specified date according to universal time.
16	getUTCMilliseconds() Returns the milliseconds in the specified date according to universal time.
17	getUTCMinutes()

	Returns the minutes in the specified date according to universal time.
18	getUTCMonth() Returns the month in the specified date according to universal time.
19	getUTCSeconds() Returns the seconds in the specified date according to universal time.
20	getYear() **Deprecated** - Returns the year in the specified date according to local time. Use getFullYear instead.
21	setDate() Sets the day of the month for a specified date according to local time.
22	setFullYear() Sets the full year for a specified date according to local time.
23	setHours() Sets the hours for a specified date according to local time.
24	setMilliseconds() Sets the milliseconds for a specified date according to local time.
25	setMinutes() Sets the minutes for a specified date according to local time.
26	setMonth() Sets the month for a specified date according to local time.
27	setSeconds() Sets the seconds for a specified date according to local time.
28	setTime() Sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC.
29	setUTCDate() Sets the day of the month for a specified date according to universal time.
30	setUTCFullYear() Sets the full year for a specified date according to universal time.
31	setUTCHours() Sets the hour for a specified date according to universal time.
32	setUTCMilliseconds() Sets the milliseconds for a specified date according to universal time.
33	setUTCMinutes()

	Sets the minutes for a specified date according to universal time.
34	setUTCMonth() Sets the month for a specified date according to universal time.
35	setUTCSeconds() Sets the seconds for a specified date according to universal time.
36	setYear() **Deprecated -** Sets the year for a specified date according to local time. Use setFullYear instead.
37	toDateString() Returns the "date" portion of the Date as a human-readable string.
38	toGMTString() **Deprecated -** Converts a date to a string, using the Internet GMT conventions. Use toUTCString instead.
39	toLocaleDateString() Returns the "date" portion of the Date as a string, using the current locale's conventions.
40	toLocaleFormat() Converts a date to a string, using a format string.
41	toLocaleString() Converts a date to a string, using the current locale's conventions.
42	toLocaleTimeString() Returns the "time" portion of the Date as a string, using the current locale's conventions.
43	toSource() Returns a string representing the source for an equivalent Date object; you can use this value to create a new object.
44	toString() Returns a string representing the specified Date object.
45	toTimeString() Returns the "time" portion of the Date as a human-readable string.
46	toUTCString() Converts a date to a string, using the universal time convention.
47	valueOf() Returns the primitive value of a Date object.

Date Static Methods

In addition to the many instance methods listed previously, the Date object also defines two static methods. These methods are invoked through the Date( ) constructor itself −

Sr.No.	Method & Description
1	Date.parse( ) Parses a string representation of a date and time and returns the internal millisecond representation of that date.
2	Date.UTC( ) Returns the millisecond representation of the specified UTC date and time.

Math Methods

Here is a list of each method and its description.

Sr.No.	Method & Description
1	abs() Returns the absolute value of a number.
2	acos() Returns the arccosine (in radians) of a number.
3	asin() Returns the arcsine (in radians) of a number.
4	atan() Returns the arctangent (in radians) of a number.
5	atan2() Returns the arctangent of the quotient of its arguments.
6	ceil() Returns the smallest integer greater than or equal to a number.
7	cos() Returns the cosine of a number.
8	exp() Returns E^N, where N is the argument, and E is Euler's constant, the base of the natural logarithm.
9	floor() Returns the largest integer less than or equal to a number.
10	log() Returns the natural logarithm (base E) of a number.
11	max() Returns the largest of zero or more numbers.
12	min() Returns the smallest of zero or more numbers.

13	pow() Returns base to the exponent power, that is, base exponent.
14	random() Returns a pseudo-random number between 0 and 1.
15	round() Returns the value of a number rounded to the nearest integer.
16	sin() Returns the sine of a number.
17	sqrt() Returns the square root of a number.
18	tan() Returns the tangent of a number.
19	toSource() Returns the string "Math".

RegExp Methods
Here is a list of each method and its description.

Sr.No.	Method & Description
1	exec() Executes a search for a match in its string parameter.
2	test() Tests for a match in its string parameter.
3	toSource() Returns an object literal representing the specified object; you can use this value to create a new object.
4	toString() Returns a string representing the specified object.

Browser Object Model:

The Browser Object Model (BOM)
There are no official standards for the **B**rowser **O**bject **M**odel (BOM).
Since modern browsers have implemented (almost) the same methods and properties for JavaScript interactivity, it is often referred to, as methods and properties of the BOM.

The Window Object
The window object is supported by all browsers. It represents the browser's window.

All global JavaScript objects, functions, and variables automatically become members of the window object.
Global variables are properties of the window object.
Global functions are methods of the window object.
Even the document object (of the HTML DOM) is a property of the window object:
window.document.getElementById("header");
is the same as:
document.getElementById("header");

Window Size
Two properties can be used to determine the size of the browser window.
Both properties return the sizes in pixels:

- window.innerHeight - the inner height of the browser window (in pixels)
- window.innerWidth - the inner width of the browser window (in pixels)

For Internet Explorer 8, 7, 6, 5:

- document.documentElement.clientHeight
- document.documentElement.clientWidth
- or
- document.body.clientHeight
- document.body.clientWidth

A practical JavaScript solution (covering all browsers):
Example

```
var w = window.innerWidth
|| document.documentElement.clientWidth
|| document.body.clientWidth;

var h = window.innerHeight
|| document.documentElement.clientHeight
|| document.body.clientHeight;
```

The example displays the browser window's height and width: (NOT including toolbars/scrollbars)

Other Window Methods
Some other methods:

- window.open() - open a new window
- window.close() - close the current window
- window.moveTo() - move the current window
- window.resizeTo() - resize the current window

Verifying forms:

Form validation normally used to occur at the server, after the client had entered all the necessary data and then pressed the Submit button. If the data entered by a client was incorrect or was simply missing, the server would have to send all the data back to the client and request that the form be resubmitted with correct information. This was really a lengthy process which used to put a lot of burden on the server.
JavaScript provides a way to validate form's data on the client's computer before sending it to the web server. Form validation generally performs two functions.

- **Basic Validation** − First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.

- **Data Format Validation** − Secondly, the data that is entered must be checked for correct form and value. Your code must include appropriate logic to test correctness of data.

Example

We will take an example to understand the process of validation. Here is a simple form in html format.

```
<html>
  <head>
    <title>Form Validation</title>
    <script type = "text/javascript">
      <!--
        // Form validation code will come here.
      //-->
    </script>
  </head>

  <body>
    <form action = "/cgi-bin/test.cgi" name = "myForm" onsubmit = "return(validate());">
      <table cellspacing = "2" cellpadding = "2" border = "1">

        <tr>
          <td align = "right">Name</td>
          <td><input type = "text" name = "Name" /></td>
        </tr>

        <tr>
          <td align = "right">EMail</td>
          <td><input type = "text" name = "EMail" /></td>
        </tr>

        <tr>
          <td align = "right">Zip Code</td>
          <td><input type = "text" name = "Zip" /></td>
        </tr>

        <tr>
          <td align = "right">Country</td>
          <td>
            <select name = "Country">
              <option value = "-1" selected>[choose yours]</option>
              <option value = "1">USA</option>
              <option value = "2">UK</option>
              <option value = "3">INDIA</option>
            </select>
          </td>
        </tr>

        <tr>
          <td align = "right"></td>
          <td><input type = "submit" value = "Submit" /></td>
        </tr>

      </table>
    </form>
  </body>
</html>
```

HTML5:

Easy Learning with HTML "Try it Yourself"

With our "Try it Yourself" editor, you can edit the HTML code and view the result:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>This is a Heading</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

Click on the "Try it Yourself" button to see how it works.

HTML Examples

In this HTML tutorial, you will find more than 200 examples. With our online "Try it Yourself" editor, you can edit and test each example yourself!

CSS3:

What is CSS?

- **CSS** stands for **C**ascading **S**tyle **S**heets
- CSS describes **how HTML elements are to be displayed on screen, paper, or in other media**
- CSS **saves a lot of work**. It can control the layout of multiple web pages all at once
- External stylesheets are stored in **CSS files**

CSS Demo - One HTML Page - Multiple Styles!

Here we will show one HTML page displayed with four different stylesheets. Click on the "Stylesheet 1", "Stylesheet 2", "Stylesheet 3", "Stylesheet 4" links below to see the different styles:

Why Use CSS?

CSS is used to define styles for your web pages, including the design, layout and variations in display for different devices and screen sizes.

CSS Example

```
body {
  background-color: lightblue;
}
```

```
h1 {
  color: white;
  text-align: center;
}

p {
  font-family: verdana;
  font-size: 20px;
}
```
CSS Solved a Big Problem
HTML was NEVER intended to contain tags for formatting a web page!
HTML was created to **describe the content** of a web page, like:
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
When tags like <font>, and color attributes were added to the HTML 3.2 specification, it started a nightmare for web developers. Development of large websites, where fonts and color information were added to every single page, became a long and expensive process.
To solve this problem, the World Wide Web Consortium (W3C) created CSS.
CSS removed the style formatting from the HTML page!
CSS Saves a Lot of Work!
The style definitions are normally saved in external .css files.
With an external stylesheet file, you can change the look of an entire website by changing just one file!

<u>HTML 5 Canvas:</u>
HTML Canvas Graphics

The HTML `<canvas>` element is used to draw graphics on a web page.
The graphic to the left is created with `<canvas>`. It shows four elements: a red rectangle, a gradient rectangle, a multicolor rectangle, and a multicolor text.

What is HTML Canvas?
The HTML `<canvas>` element is used to draw graphics, on the fly, via JavaScript.
The `<canvas>` element is only a container for graphics. You must use JavaScript to actually draw the graphics.
Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

Browser Support
The numbers in the table specify the first browser version that fully supports the `<canvas>` element.

Property					
frames	Yes	Yes	Yes	Yes	Yes

Canvas Examples
A canvas is a rectangular area on an HTML page. By default, a canvas has no border and no content.
The markup looks like this:
```
<canvas id="myCanvas" width="200" height="100"></canvas>
```

Note: Always specify an id attribute (to be referred to in a script), and a width and height attribute to define the size of the canvas. To add a border, use the style attribute.

Here is an example of a basic, empty canvas:

Example

```
<canvas id="myCanvas" width="200" height="100" style="border:1px solid #000000;">
</canvas>
```

Add a JavaScript

After creating the rectangular canvas area, you must add a JavaScript to do the drawing.

Here are some examples:

Draw a Line

Example

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.moveTo(0, 0);
ctx.lineTo(200, 100);
ctx.stroke();
</script>
```

Draw a Circle

Example

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(95, 50, 40, 0, 2 * Math.PI);
ctx.stroke();
</script>
```

Draw a Text

Example

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.font = "30px Arial";
ctx.fillText("Hello World", 10, 50);
</script>
```

Stroke Text

Example

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.font = "30px Arial";
ctx.strokeText("Hello World", 10, 50);
</script>
```

Draw Linear Gradient

Example

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");

// Create gradient
var grd = ctx.createLinearGradient(0, 0, 200, 0);
grd.addColorStop(0, "red");
grd.addColorStop(1, "white");

// Fill with gradient
ctx.fillStyle = grd;
ctx.fillRect(10, 10, 150, 80);
</script>
```

Draw Circular Gradient

Example

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");

// Create gradient
var grd = ctx.createRadialGradient(75, 50, 5, 90, 60, 100);
grd.addColorStop(0, "red");
grd.addColorStop(1, "white");

// Fill with gradient
ctx.fillStyle = grd;
ctx.fillRect(10, 10, 150, 80);
</script>
```

Draw Image

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
var img = document.getElementById("scream");
ctx.drawImage(img, 10, 10);
</script>
```

Website creation using Tools:

List Of The Top Web Development Tools

Enlisted below are the most popular tools for Web Development that are used worldwide.

1. Angular.JS
2. Chrome DevTools
3. Sass
4. Grunt
5. CodePen
6. TypeScript
7. GitHub

8. NPM
9. JQuery
10. Bootstrap
11. Visual Studio Code
12. Sublime Text
13. Sketch

Unit-II

Java

Introduction to object-oriented programming-Features of Java – Data types, variables and arrays –Operators – Control statements – Classes and Methods – Inheritance. Packages and Interfaces –
Exception Handling – Multithreaded Programming – Input/Output – Files – Utility Classes – StringHandling.

Introduction to Object Oriented programming:

OOP stands for **Object-Oriented Programming**.
Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.
Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.
Java - What are Classes and Objects?
Classes and objects are the two main aspects of object-oriented programming.
Look at the following illustration to see the difference between class and objects:

class
Fruit
objects
Apple
Banana
Mango

Another example:

class
Car
objects
Volvo
Audi
Toyota

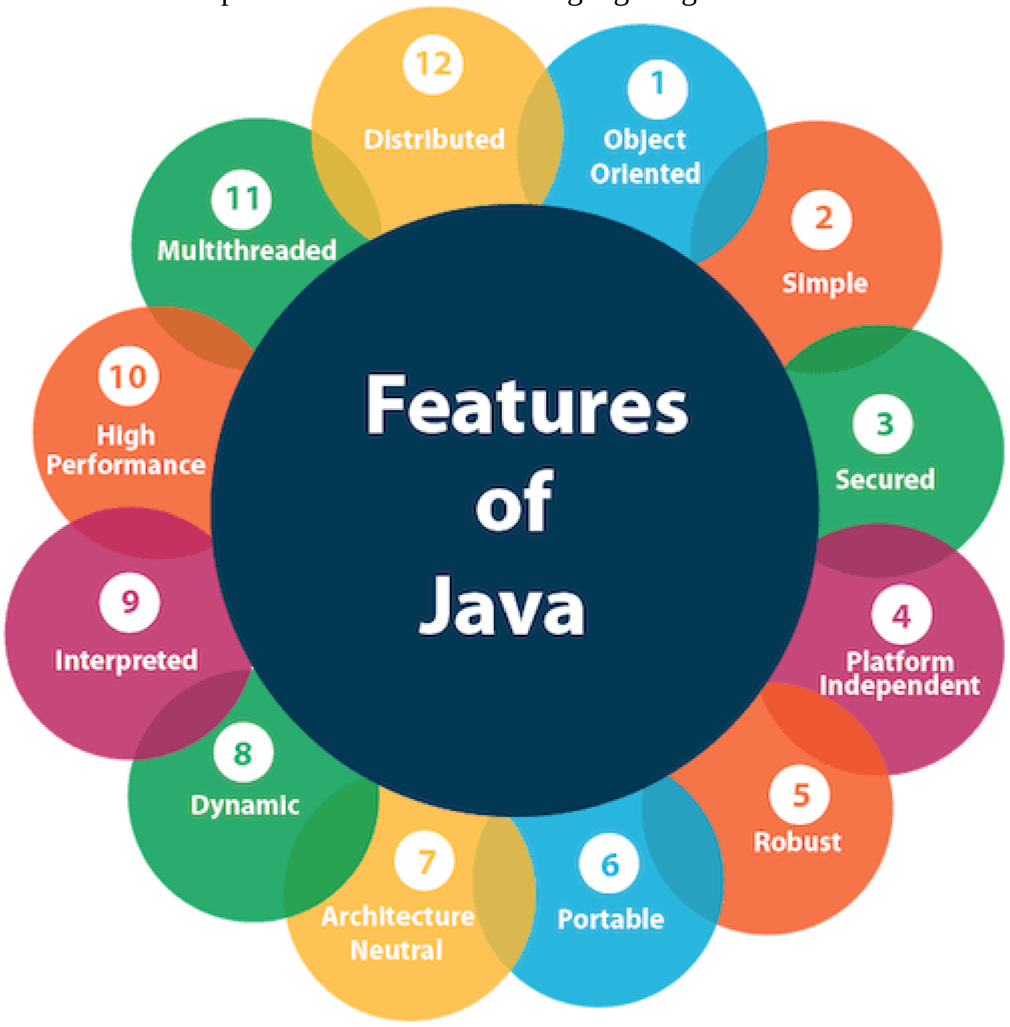
So, a class is a template for objects, and an object is an instance of a class.
When the individual objects are created, they inherit all the variables and methods from the class.
You will learn much more about classes and objects in the next chapter.

Features of Java:

Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent

features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*.
A list of most important features of Java language is given below.



1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

Java Data Types:

As explained in the previous chapter, a variable in Java must be a specified data type:

Example

```
int myNum = 5;             // Integer (whole number)
float myFloatNum = 5.99f;    // Floating point number
char myLetter = 'D';         // Character
```

```
boolean myBool = true;       // Boolean
String myText = "Hello";     // String
```

Data types are divided into two groups:

- Primitive data types - includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter)

Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Primitive number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: float and double.

Integer Types

Byte

The byte data type can store whole numbers from -128 to 127. This can be used instead of int or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
byte myNum = 100;
System.out.println(myNum);
```

Short

The short data type can store whole numbers from -32768 to 32767:

Example

```
short myNum = 5000;
System.out.println(myNum);
```

Int

The int data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

Example
```
int myNum = 100000;
System.out.println(myNum);
```
Long

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

Example
```
long myNum = 15000000000L;
System.out.println(myNum);
```
Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The float data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you should end the value with an "f":

Example
```
float myNum = 5.75f;
System.out.println(myNum)
```
Double

The double data type can store fractional numbers from 1.7e−308 to 1.7e+308. Note that you should end the value with a "d":

Example
```
double myNum = 19.99d;
System.out.println(myNum);
```
Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example
```
float f1 = 35e3f;
double d1 = 12E4d;
System.out.println(f1);
System.out.println(d1);
```
Booleans

A boolean data type is declared with the boolean keyword and can only take the values true or false:

Example
```
boolean isJavaFun = true;
boolean isFishTasty = false;
System.out.println(isJavaFun);     // Outputs true
System.out.println(isFishTasty);   // Outputs false
```
Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

Characters

The char data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

Example
```
char myGrade = 'B';
System.out.println(myGrade);
```
Alternatively, you can use ASCII values to display certain characters:

Example
```
char a = 65, b = 66, c = 67;
System.out.println(a);
System.out.println(b);
System.out.println(c);
```
Strings
The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:
Example
```
String greeting = "Hello World";
System.out.println(greeting);
```
Non-Primitive Data Types
Non-primitive data types are called **reference types** because they refer to objects.
The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc. You will learn more about these in a later chapter.

Variables and arrays:

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
You must declare all variables before they can be used. Following is the basic form of a variable declaration −
data type variable [= value][, variable [= value] ...] ;
Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.
Following are valid examples of variable declaration and initialization in Java −
Example
```
int a, b, c;         // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';        // the char variable a iis initialized with value 'a'
```
This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java −

- Local variables
- Instance variables
- Class/Static variables

Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
public class Test {
   public void pupAge() {
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]) {
      Test test = new Test();
      test.pupAge();
   }
}
```

This will produce the following result −

Output

```
Puppy age is: 7
```

Example

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test {
   public void pupAge() {
      int age;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]) {
      Test test = new Test();
      test.pupAge();
   }
}
```

This will produce the following error while compiling it −

Output

```
Test.java:4:variable number might not have been initialized
age = age + 7;
         ^
1 error
```

Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

Example

```
import java.io.*;
public class Employee {

  // this instance variable is visible for any child class.
  public String name;

  // salary  variable is visible in Employee class only.
  private double salary;

  // The name variable is assigned in the constructor.
  public Employee (String empName) {
    name = empName;
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal) {
    salary = empSal;
  }

  // This method prints the employee details.
  public void printEmp() {
    System.out.println("name  : " + name );
    System.out.println("salary :" + salary);
  }

  public static void main(String args[]) {
    Employee empOne = new Employee("Ransika");
```

```
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

This will produce the following result −

Output

```
name  : Ransika
salary :1000.0
```

Class/Static Variables

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

Example

```
import java.io.*;
public class Employee {

   // salary  variable is a private static variable
   private static double salary;

   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";

   public static void main(String args[]) {
      salary = 1000;
      System.out.println(DEPARTMENT + "average salary:" + salary);
   }
}
```

This will produce the following result −

Output

```
Development average salary:1000
```

Java – Arrays:

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

dataType[] arrayRefVar;   // preferred way.

or

dataType arrayRefVar[]; // works but not preferred way.

Note − The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax −

```
double[] myList;   // preferred way.
or
double myList[];   // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax −

Syntax

arrayRefVar = new dataType[arraySize];

The above statement does two things −

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows −

dataType[] arrayRefVar = {value0, value1, ..., valuek};

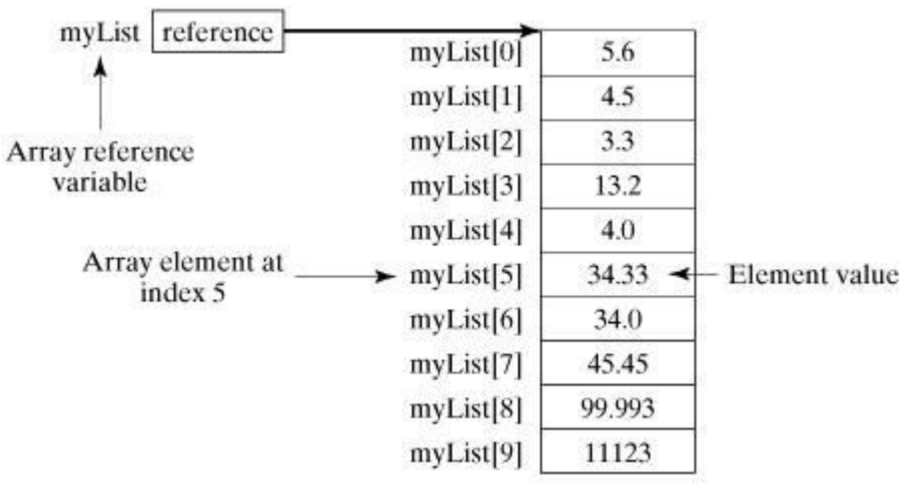
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays −

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");
      }

      // Summing all elements
      double total = 0;
      for (int i = 0; i < myList.length; i++) {
         total += myList[i];
      }
      System.out.println("Total is " + total);

      // Finding the largest element
      double max = myList[0];
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) max = myList[i];
      }
      System.out.println("Max is " + max);
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
```

```
Total is 11.7
Max is 3.5
```

The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example

The following code displays all the elements in the array myList −

```java
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (double element: myList) {
         System.out.println(element);
      }
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
```

Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array −

Example

```java
public static void printArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + " ");
   }
}
```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 −

Example

```java
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array −

Example

```java
public static int[] reverse(int[] list) {
   int[] result = new int[list.length];

   for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
      result[j] = list[i];
   }
   return result;
}
```

The Arrays Class

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

Sr.No.	Method & Description
1	**public static int binarySearch(Object[] a, Object key)** Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns (–(insertion point + 1)).
2	**public static boolean equals(long[] a, long[] a2)** Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)
3	**public static void fill(int[] a, int val)** Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.)
4	**public static void sort(Object[] a)** Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)

Java - Basic Operators:

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups −

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators −

Assume integer variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.

<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

a = 0011 1100
b = 0000 1101
----
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

The following table lists the bitwise operators −

Assume integer variable A holds 60 and variable B holds 13 then −

Show Examples

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators −

Assume Boolean variables A holds true and variable B holds false, then −

Show Examples

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language −

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as −

variable x = (expression) ? value if true : value if false

Following is an example −

Example

```
public class Test {

   public static void main(String args[]) {
      int a, b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

This will produce the following result −

Output

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as −

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example −

Example

```
public class Test {

   public static void main(String args[]) {

      String name = "James";

      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

This will produce the following result −

Output

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example −

Example

```
class Vehicle {}

public class Car extends Vehicle {

   public static void main(String args[]) {

      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

This will produce the following result −

Output

```
true
```

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3 * 2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	expression++ expression--	Left to right
Unary	++expression —-expression +expression –expression ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= ^= |= <<= >>= >>>=	Right to left

Control statements:

Decision Making in Java (if, if-else, switch, break, continue, jump)

Decision Making in programming is similar to decision making in real life. In programming also we face some situations where we want a certain block of code to be executed when some condition is fulfilled.
A programming language uses control statements to control the flow of execution of program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.
Java’s Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

These statements allow you to control the flow of your program’s execution based upon conditions known only during run time.

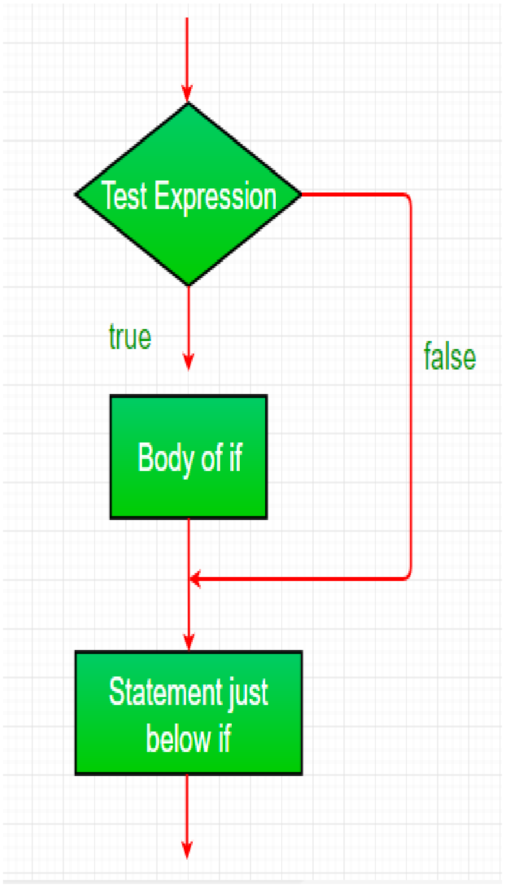
- **if**: if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain

condition is true then a block of statement is executed otherwise not.
Syntax:

```
• if(condition)
• {
•    // Statements to execute if
•    // condition is true
• }
```

Here, **condition** after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements under it.
If we do not provide the curly braces '{' and '}' after **if(condition)** then by default if statement will consider the immediate one statement to be inside its block. For example,

```
if(condition)
   statement1;
   statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

Flow chart:


Example:
filter_none
edit
play_arrow
brightness_4

```
// Java program to illustrate If statement
class IfDemo
```

```
{
    public static void main(String args[])
    {
        int i = 10;

        if (i > 15)
            System.out.println("10 is less than 15");

        // This statement will be executed
        // as if considers one statement by default
        System.out.println("I am Not in if");
    }
}
```

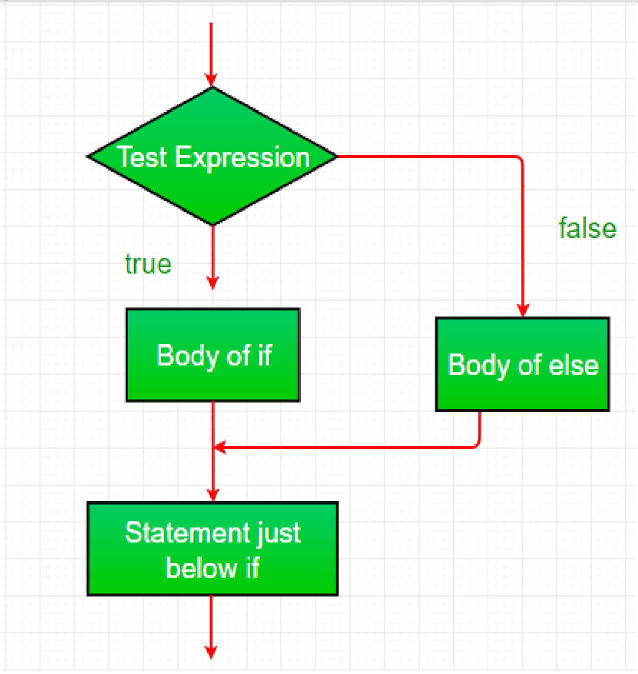
Output:

```
I am Not in if
```

- **if-else**: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

  Syntax:

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```



Example:
filter_none
edit
play_arrow
brightness_4

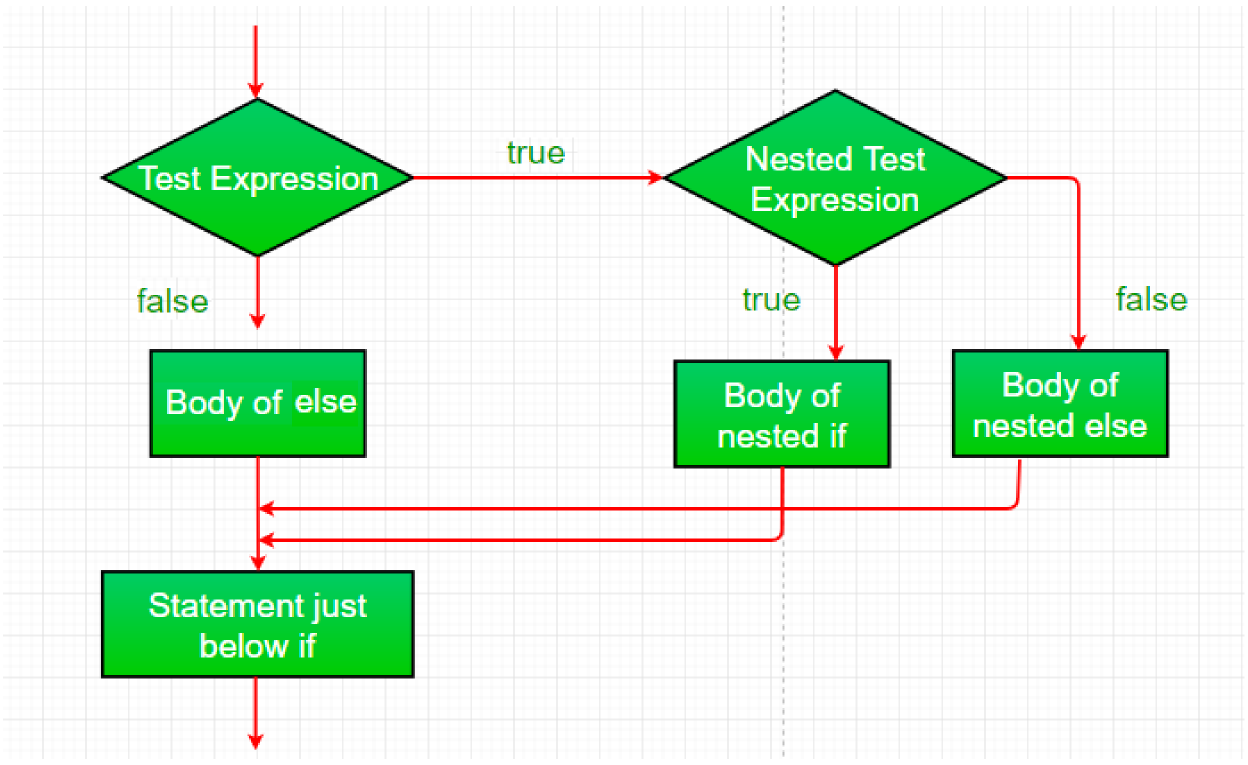
```
// Java program to illustrate if-else statement
class IfElseDemo
{
    public static void main(String args[])
    {
        int i = 10;

        if (i < 15)
            System.out.println("i is smaller than 15");
        else
            System.out.println("i is greater than 15");
    }
}
```

Output:

```
i is smaller than 15
```

- **nested-if:** A nested if is an if statement that is the target of another if or else. Nested if statements means an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.
  Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```



Example:
filter_none
edit
play_arrow
brightness_4

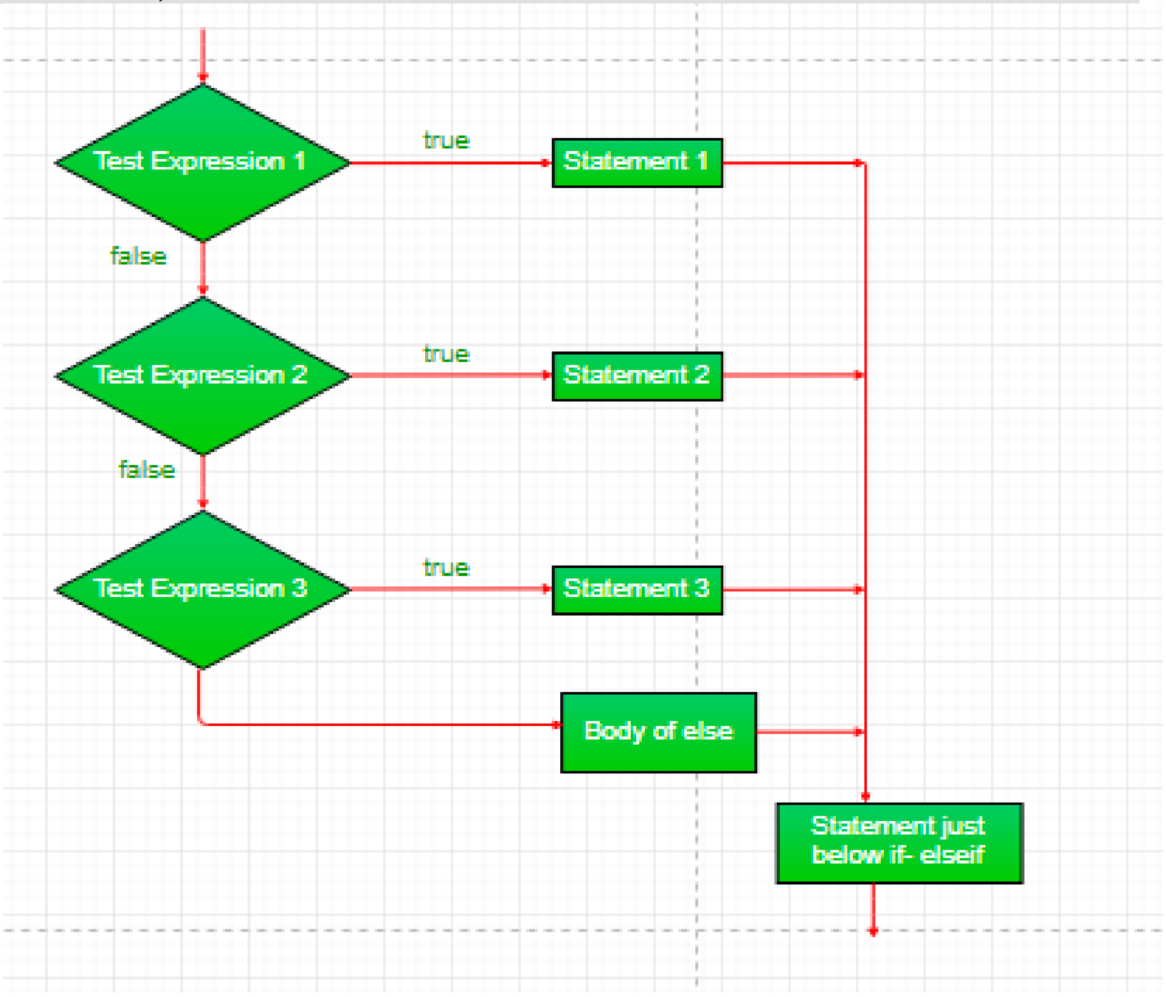
```
// Java program to illustrate nested-if statement
class NestedIfDemo
{
    public static void main(String args[])
    {
        int i = 10;

        if (i == 10)
        {
            // First if statement
            if (i < 15)
                System.out.println("i is smaller than 15");

            // Nested - if statement
            // Will only be executed if statement above
            // it is true
            if (i < 12)
                System.out.println("i is smaller than 12 too");
            else
                System.out.println("i is greater than 15");
        }
    }
}
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

- **if-else-if ladder:**
  Here, a user can decide among multiple options.The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```



Example:

filter_none
edit

play_arrow
brightness_4

```
// Java program to illustrate if-else-if ladder
class ifelseifDemo
{
    public static void main(String args[])
    {
        int i = 20;

        if (i == 10)
            System.out.println("i is 10");
        else if (i == 15)
            System.out.println("i is 15");
        else if (i == 20)
            System.out.println("i is 20");
        else
            System.out.println("i is not present");
    }
}
```

Output:

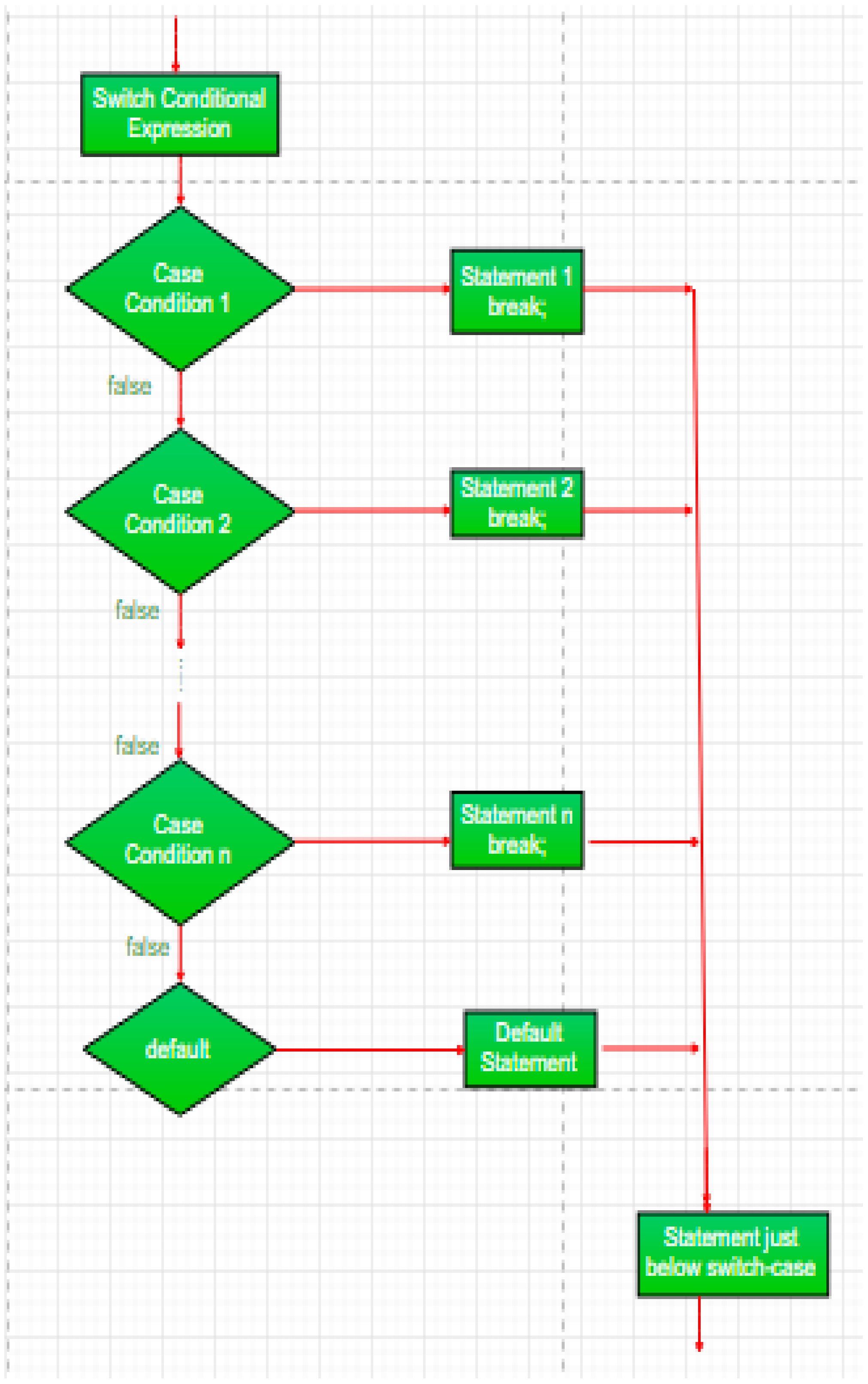
```
i is 20
```

switch-case

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
Syntax:

```
• switch (expression)
• {
•   case value1:
•     statement1;
•     break;
•   case value2:
•     statement2;
•     break;
•   .
•   .
•   case valueN:
•     statementN;
•     break;
•   default:
•     statementDefault;
}
```

- Expression can be of type byte, short, int char or an enumeration. Beginning with JDK7, *expression* can also be of type String.
- Dulplicate case values are not allowed.
- The default statement is optional.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If omitted, execution will continue on into the next case.


Switch Conditional Expression
Case Condition 1
Statement 1 break;
false
Case Condition 2
Statement 2 break;
false
false
Case Condition n
Statement n break;
false
default
Default Statement
Statement just below switch-case

Example:

filter_none
edit
play_arrow
brightness_4

```
// Java program to illustrate switch-case
class SwitchCaseDemo
{
    public static void main(String args[])
    {
        int i = 9;
        switch (i)
        {
        case 0:
            System.out.println("i is zero.");
            break;
        case 1:
            System.out.println("i is one.");
            break;
        case 2:
            System.out.println("i is two.");
            break;
        default:
            System.out.println("i is greater than 2.");
        }
    }
}
```

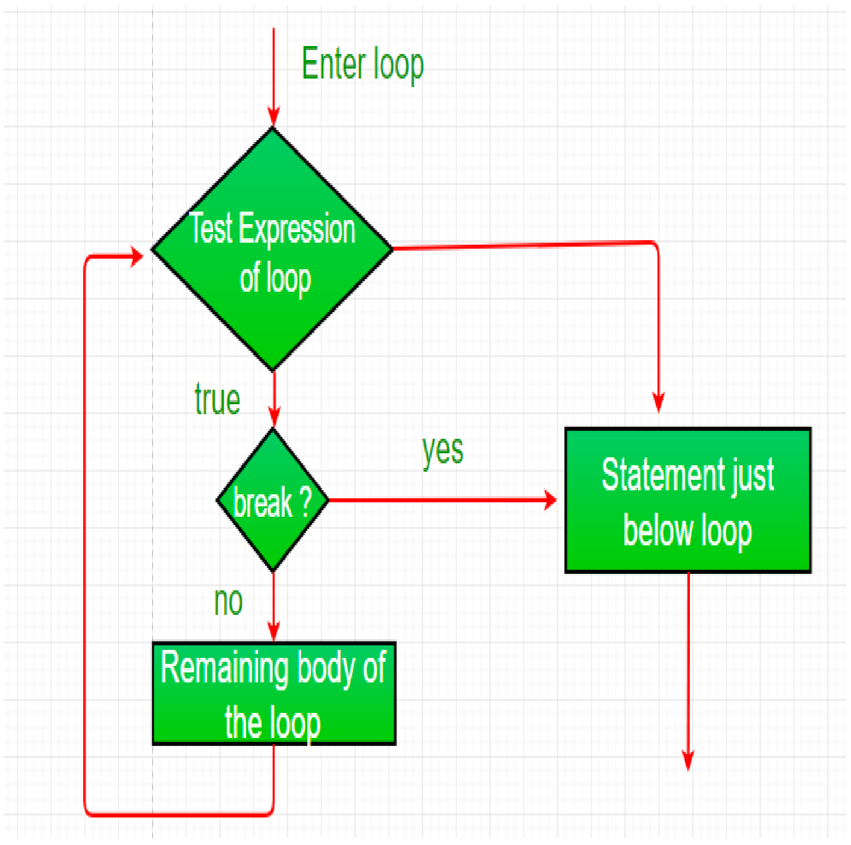
Output:

```
i is greater than 2.
```

-
- **jump:** Java supports three jump statement: **break, continue** and **return**. These three statements transfer control to other part of the program.
    - **Break:** In Java, break is majorly used for:
        - Terminate a sequence in a switch statement (discussed above).
        - To exit a loop.
        - Used as a “civilized” form of goto.

    Using break to exit a Loop

    Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
    Note: Break, when used inside a set of nested loops, will only break out of the innermost loop.



Example:

filter_none
edit
play_arrow
brightness_4

```
// Java program to illustrate using
// break to exit a loop
class BreakLoopDemo
{
    public static void main(String args[])
    {
        // Initially loop is set to run from 0-9
        for (int i = 0; i < 10; i++)
        {
            // terminate loop when i is 5.
            if (i == 5)
                break;

            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

Output:

```
i: 0
i: 1
```

```
i: 2
i: 3
i: 4
Loop complete.
```

Using break as a Form of Goto

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. Java uses label. A Label is use to identifies a block of code.

Syntax:

```
label:
{
  statement1;
  statement2;
  statement3;
  .
  .
}
```

Now, break statement can be use to jump out of target block.

Note: You cannot break to any label which is not defined for an enclosing block.

Syntax:

```
break label;
```

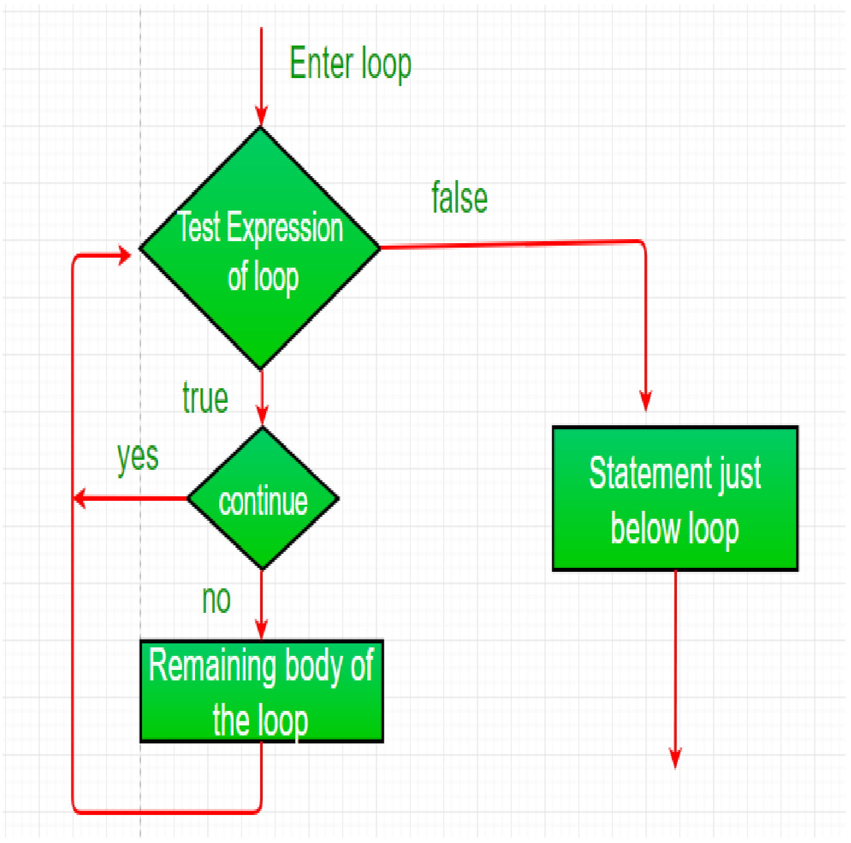
Example:

filter_none
edit
play_arrow
brightness_4

```
// Java program to illustrate using break with goto
class BreakLabelDemo
{
    public static void main(String args[])
    {
        boolean t = true;

        // label first
        first:
        {
            // Illegal statement here as label second is not
            // introduced yet break second;
            second:
            {
                third:
                {
                    // Before break
                    System.out.println("Before the break statement");

                    // break will take the control out of
                    // second label
                    if (t)
                        break second;
                    System.out.println("This won't execute.");
                }
                System.out.println("This won't execute.");
            }

            // First block
            System.out.println("This is after second block.");
        }
    }
}
```
Output:

```
Before the break.
This is after second block.
```

- **Continue:** Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.



Example:

filter_none
edit
play_arrow
brightness_4

```
// Java program to illustrate using
// continue in an if statement
class ContinueDemo
{
    public static void main(String args[])
    {
        for (int i = 0; i < 10; i++)
        {
            // If the number is even
            // skip and continue
            if (i%2 == 0)
                continue;

            // If number is odd, print it
            System.out.print(i + " ");
        }
    }
}
```

Output:

```
1 3 5 7 9
```

Classes and Methods:

Java Class Methods

You learned from the Java Methods chapter that methods are declared within a class, and that they are used to perform certain actions:

Example

Create a method named myMethod() in MyClass:

```java
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

myMethod() prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

Example

Inside main, call myMethod():

```java
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

Static vs. Non-Static

You will often see Java programs that have either static or public attributes and methods.

In the example above, we created a static method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

Example

An example to demonstrate the differences between static and public **methods**:

```
public class MyClass {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an error

    MyClass myObj = new MyClass(); // Create an object of MyClass
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

Access Methods With an Object

Example

Create a Car object named myCar. Call the fullThrottle() and speed() methods on the myCar object, and run the program:

```
// Create a Car class
public class Car {

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Car myCar = new Car();     // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}

// The car is going as fast as it can!
// Max speed is: 200
```

Example explained
1) We created a custom Car class with the class keyword.
2) We created the fullThrottle() and speed() methods in the Car class.
3) The fullThrottle() method and the speed() method will print out some text, when they are called.
4) The speed() method accepts an int parameter called maxSpeed - we will use this in **8)**.
5) In order to use the Car class and its methods, we need to create an **object** of the Car Class.
6) Then, go to the main() method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).
7) By using the new keyword we created a Car object with the name myCar.
8) Then, we call the fullThrottle() and speed() methods on the myCar object, and run the program using the name of the object (myCar), followed by a dot (.), followed by the name of the method (fullThrottle(); and speed(200);). Notice that we add an int parameter of **200** inside the speed() method.
Using Multiple Classes
Like we specified in the Classes chapter, it is a good practice to create an object of a class and access it in another class.
Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

- Car.java
- OtherClass.java

Car.java
```
public class Car {
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
}
```
OtherClass.java
```
class OtherClass {
  public static void main(String[] args) {
    Car myCar = new Car();     // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}
```
When both files have been compiled:
```
C:\Users\Your Name>javac Car.java
C:\Users\Your Name>javac OtherClass.java
```
Run the OtherClass.java file:
```
C:\Users\Your Name>java OtherClass
```
And the output will be:
```
The car is going as fast as it can!
Max speed is: 200
```

Inheritance:

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.
The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).
extends Keyword
extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.
Syntax

```
class Super {
  .....
  .....
}
class Sub extends Super {
  .....
  .....
}
```

Sample Code
Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.
Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.
Copy and paste the following program in a file with name My_Calculation.java
Example

```
class Calculation {
  int z;

  public void addition(int x, int y) {
    z = x + y;
    System.out.println("The sum of the given numbers:"+z);
  }

  public void Subtraction(int x, int y) {
    z = x - y;
    System.out.println("The difference between the given numbers:"+z);
  }
}

public class My_Calculation extends Calculation {
  public void multiplication(int x, int y) {
    z = x * y;
    System.out.println("The product of the given numbers:"+z);
  }

  public static void main(String args[]) {
    int a = 20, b = 10;
    My_Calculation demo = new My_Calculation();
```

```
      demo.addition(a, b);
      demo.Subtraction(a, b);
      demo.multiplication(a, b);
   }
}
```

Compile and execute the above code as shown below.

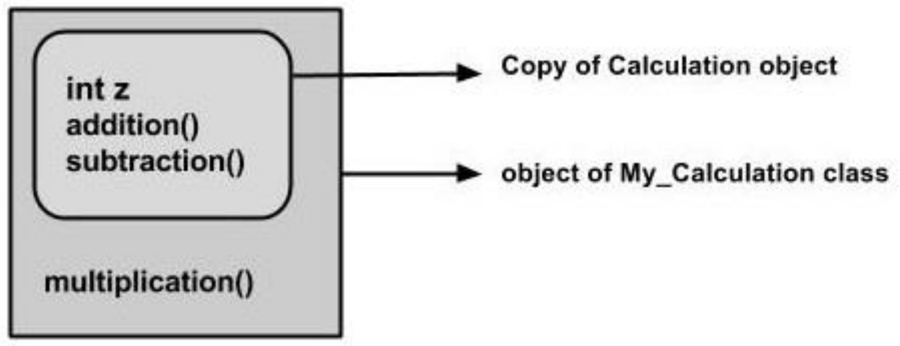
```
javac My_Calculation.java
java My_Calculation
```

After executing the program, it will produce the following result −

Output

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass My_Calculation.

```
Calculation demo = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);
```

Note − A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

super.variable
super.method();

Sample Code

This section provides you a program that demonstrates the usage of the **super** keyword.
In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.
Copy and paste the program in a file with name Sub_class.java.

Example

```
class Super_class {
   int num = 20;

   // display method of superclass
   public void display() {
      System.out.println("This is the display method of superclass");
   }
}

public class Sub_class extends Super_class {
   int num = 10;

   // display method of sub class
   public void display() {
      System.out.println("This is the display method of subclass");
   }

   public void my_method() {
      // Instantiating subclass
      Sub_class sub = new Sub_class();

      // Invoking the display() method of sub class
      sub.display();

      // Invoking the display() method of superclass
      super.display();

      // printing the value of variable num of subclass
      System.out.println("value of the variable named num in sub class:"+ sub.num);

      // printing the value of variable num of superclass
      System.out.println("value of the variable named num in super class:"+ super.num);
   }

   public static void main(String args[]) {
      Sub_class obj = new Sub_class();
      obj.my_method();
   }
}
```

Compile and execute the above code using the following syntax.

```
javac Super_Demo
java Super
```

On executing the program, you will get the following result −

Output

```
This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20
```

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

Example

```
class Superclass {
   int age;

   Superclass(int age) {
      this.age = age;
   }

   public void getAge() {
      System.out.println("The value of the variable named age in super class is: " +age);
   }
}

public class Subclass extends Superclass {
   Subclass(int age) {
      super(age);
   }

   public static void main(String args[]) {
      Subclass s = new Subclass(24);
      s.getAge();
   }
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass
```

java Subclass

On executing the program, you will get the following result −

Output

```
The value of the variable named age in super class is: 24
```

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal {
}

public class Mammal extends Animal {
}

public class Reptile extends Animal {
}

public class Dog extends Mammal {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true −

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say −

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

```
class Animal {
}

class Mammal extends Animal {
}

class Reptile extends Animal {
}

public class Dog extends Mammal {

   public static void main(String args[]) {
      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This will produce the following result −

Output

```
true
true
true
```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {
}

public class Mammal implements Animal {
}

public class Dog extends Mammal {
}
```

The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

```
interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {

   public static void main(String args[]) {
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This will produce the following result −

Output

```
true
true
true
```

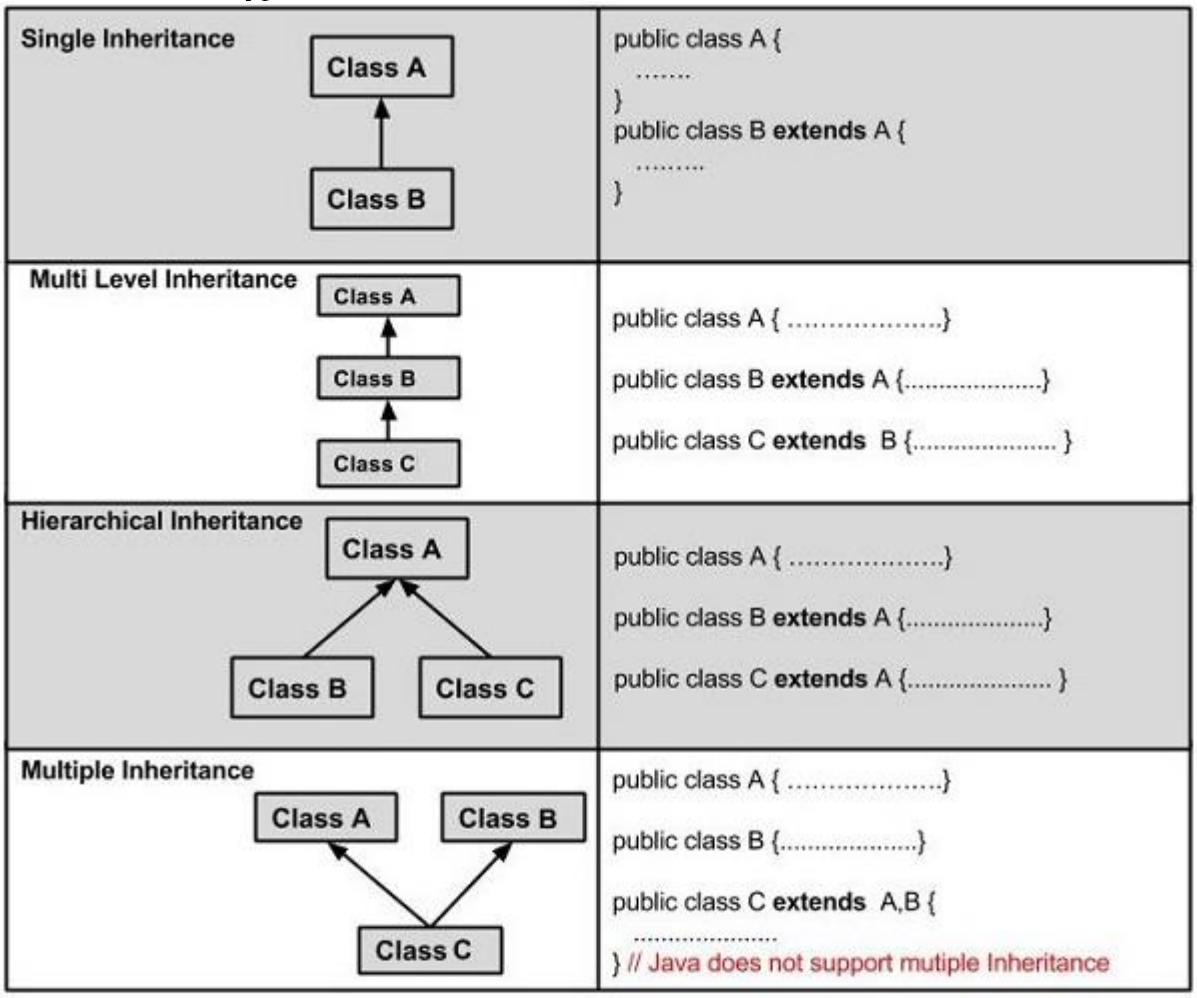
HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.
Lets look into an example −
Example

```
public class Vehicle{}
public class Speed{}

public class Van extends Vehicle {
  private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.
In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

Types of Inheritance
There are various types of inheritance as demonstrated below.



```
public class A {
  .......
}
public class B extends A {
  .........
}
```

```
public class A { ...................}

public class B extends A {....................}

public class C extends  B {..................... }
```

```
public class A { ...................}

public class B extends A {....................}

public class C extends A {..................... }
```

```
public class A { ...................}

public class B {....................}

public class C extends  A,B {
  .....................
} // Java does not support mutiple Inheritance
```

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal −

Example

```
public class extends Animal, Mammal{}
```

Packages and Interfaces:

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are −

- **java.lang** − bundles the fundamental classes
- **java.io** − classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* −

```
/* File name : Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

Now, let us implement the above interface in the same package *animals* −

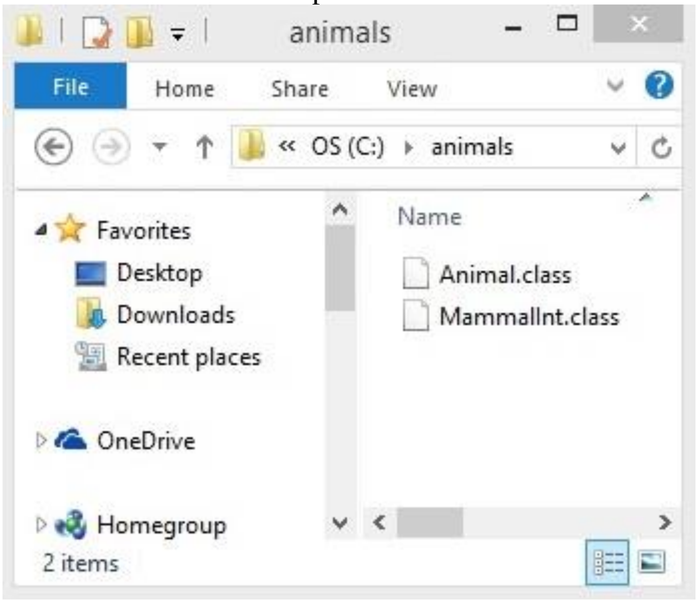
```
package animals;
/* File name : MammalInt.java */
```

```
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

Now compile the java files as shown below −

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.



You can execute the class file within the package and get the result as shown below.

```
Mammal eats
Mammal travels
```

The import Keyword

If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.

Example

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;
public class Boss {
  public void payEmployee(Employee e) {
    e.mailCheck();
  }
}
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example −

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example −

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example −

```
import payroll.Employee;
```

Note − A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages

Two major results occur when a class is placed in a package −

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java −

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**.

For example −

```
// File Name :  Car.java
package vehicle;

public class Car {
  // Class implementation.
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs −

....\vehicle\Car.java

Now, the qualified class name and pathname would be as follows −

- Class name → vehicle.Car
- Path name → vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names.

Example − A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example − The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this −

```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**.

For example −

```
// File Name: Dell.java
package com.apple.computers;

public class Dell {
}

class Ups {
}
```

Now, compile this file as follows using -d option −

```
$javac -d . Dell.java
```

The files will be compiled as follows −

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in *\com\apple\computers\* as follows −

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as −

```
<path-one>\sources\com\apple\computers\Dell.java

<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give access to the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\computers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell) −

- In Windows → C:\> set CLASSPATH
- In UNIX → % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use −

- In Windows → C:\> set CLASSPATH =

- In UNIX → % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable −

- In Windows → set CLASSPATH = C:\users\jack\java\classes
- In UNIX → % CLASSPATH = /home/jack/java/classes; export CLASSPATH

Interfaces:

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

Following is an example of an interface −

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
  // Any number of final, static fields
  // Any number of abstract method declarations\
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.

- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }

  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

This will produce the following result −

Output

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed −

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

```
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

Example

```
package java.util;
public interface EventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

Creates a common parent − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

Exception Handling:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

  public static void main(String args[]) {
    File file = new File("E://file.txt");
    FileReader fr = new FileReader(file);
  }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be
caught or declared to be thrown
    FileReader fr = new FileReader(file);
                    ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

```
public class Unchecked_Demo {

  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
  }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
          at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

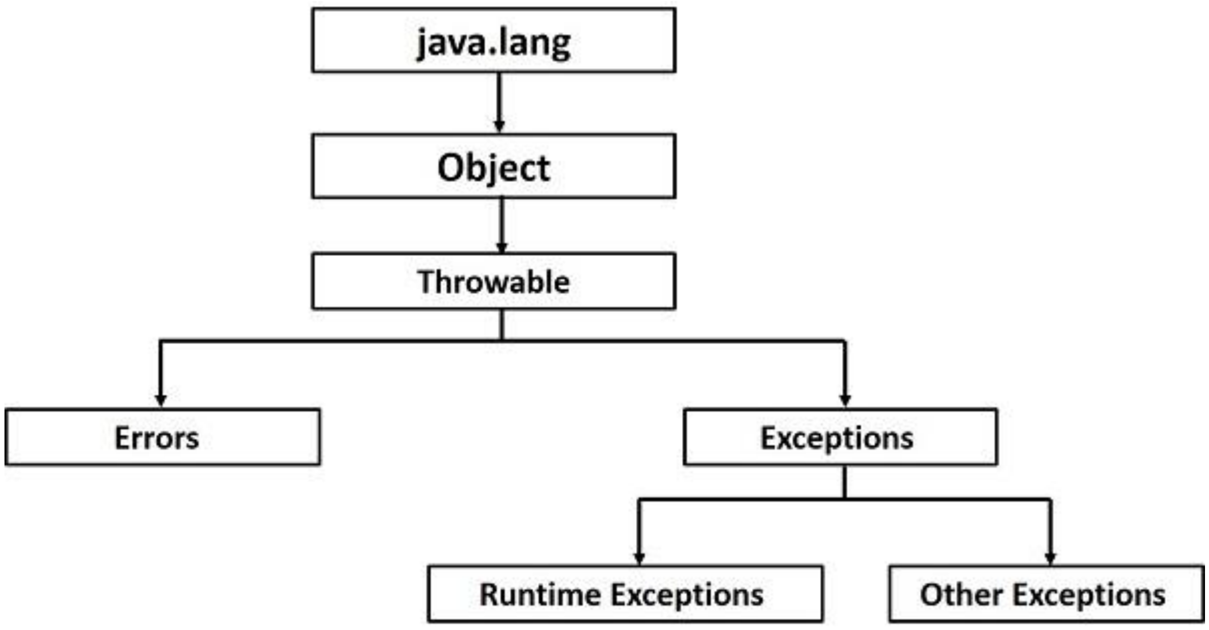
- **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Following is a list of most common checked and unchecked Java's Built-in Exceptions.
Exceptions Methods
Following is the list of important methods available in the Throwable class.

Sr.No.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

Syntax

```
try {
   // Protected code
} catch (ExceptionName e1) {
   // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;

public class ExcepTest {

   public static void main(String args[]) {
      try {
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try {
   file = new FileInputStream(fileName);
   x = (byte) file.read();
} catch (IOException i) {
   i.printStackTrace();
   return -1;
} catch (FileNotFoundException f) // Not valid! {
   f.printStackTrace();
   return -1;
}
```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it −

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
   throw ex;
```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException −

Example

```
import java.io.*;
public class className {

   public void deposit(double amount) throws RemoteException {
      // Method implementation
      throw new RemoteException();
   }
   // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException −

Example

```
import java.io.*;
public class className {

   public void withdraw(double amount) throws RemoteException,
      InsufficientFundsException {
      // Method implementation
   }
   // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

Syntax

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}finally {
   // The finally block always executes.
}
```

Example

```
public class ExcepTest {

   public static void main(String args[]) {
      int a[] = new int[2];
      try {
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }finally {
         a[0] = 6;
         System.out.println("First element value: " + a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following −

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

The try-with-resources

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

Example

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {

   public static void main(String args[]) {
      FileReader fr = null;
      try {
         File file = new File("file.txt");
         fr = new FileReader(file); char [] a = new char[50];
         fr.read(a);   // reads the content to the array
         for(char c : a)
         System.out.print(c);   // prints the characters one by one
      } catch (IOException e) {
         e.printStackTrace();
      }finally {
         try {
            fr.close();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

try-with-resources, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

Syntax

```
try(FileReader fr = new FileReader("file path")) {
   // use the resource
   } catch () {
      // body of catch
```

```
  }
}
```

Following is the program that reads the data in a file using try-with-resources statement.

Example

```
import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

   public static void main(String args[]) {
      try(FileReader fr = new FileReader("E://file.txt")) {
         char [] a = new char[50];
         fr.read(a);   // reads the contentto the array
         for(char c : a)
         System.out.print(c);   // prints the characters one by one
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Following points are to be kept in mind while working with try-with-resources statement.

- To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
- You can declare more than one class in try-with-resources statement.
- While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block.
- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes −

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below −

```
class MyException extends Exception {
}
```

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example

```
// File Name InsufficientFundsException.java
```

```java
import java.io.*;

public class InsufficientFundsException extends Exception {
  private double amount;

  public InsufficientFundsException(double amount) {
    this.amount = amount;
  }

  public double getAmount() {
    return amount;
  }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```java
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount {
  private double balance;
  private int number;

  public CheckingAccount(int number) {
    this.number = number;
  }

  public void deposit(double amount) {
    balance += amount;
  }

  public void withdraw(double amount) throws InsufficientFundsException {
    if(amount <= balance) {
      balance -= amount;
    }else {
      double needs = amount - balance;
      throw new InsufficientFundsException(needs);
    }
  }

  public double getBalance() {
    return balance;
  }

  public int getNumber() {
    return number;
  }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```java
// File Name BankDemo.java
public class BankDemo {

  public static void main(String [] args) {
    CheckingAccount c = new CheckingAccount(101);
    System.out.println("Depositing $500...");
    c.deposit(500.00);

    try {
      System.out.println("\nWithdrawing $100...");
      c.withdraw(100.00);
      System.out.println("\nWithdrawing $600...");
      c.withdraw(600.00);
    } catch (InsufficientFundsException e) {
      System.out.println("Sorry, but you are short $" + e.getAmount());
      e.printStackTrace();
    }
  }
}
```

Compile all the above three files and run BankDemo. This will produce the following result –

Output

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

Common Exceptions

In Java, it is possible to define two catergories of Exceptions and Errors.

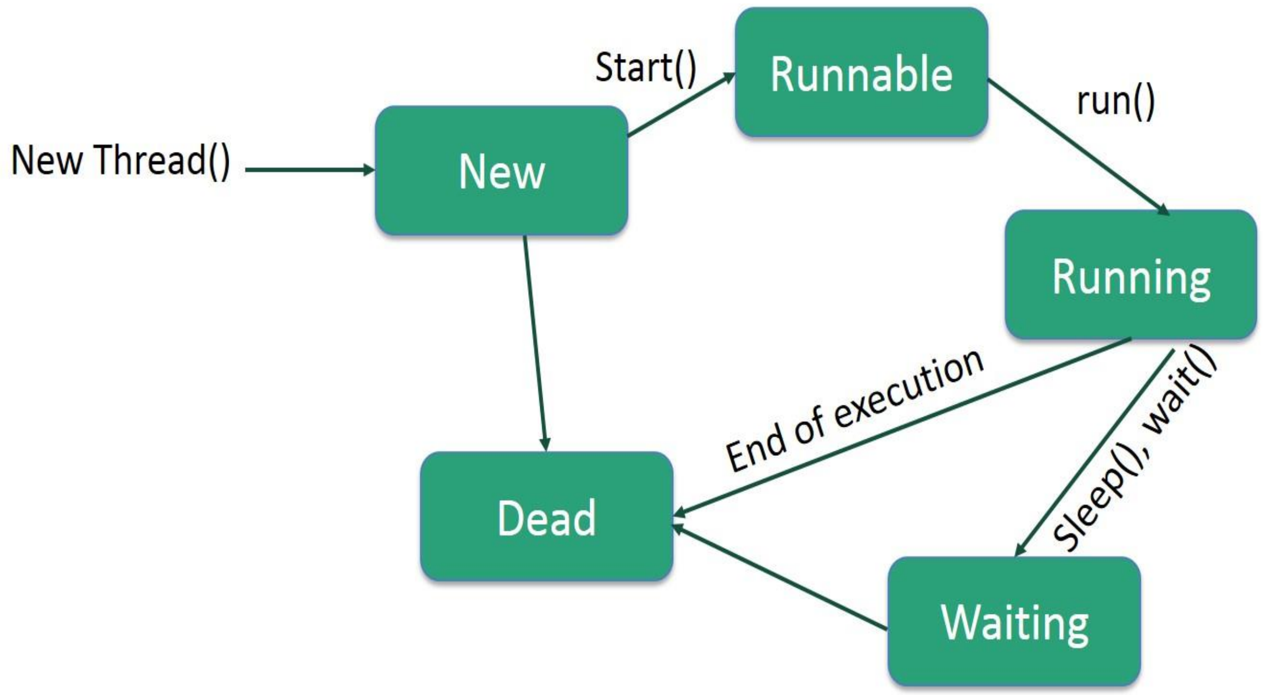
- **JVM Exceptions** – These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.
- **Programmatic Exceptions** – These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

Multithreaded Programming:

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of

the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.
Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.
Life Cycle of a Thread
A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle −

- **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities
Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic steps −

Step 1

As a first step, you need to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method −

public void run( )

Step 2

As a second step, you will instantiate a **Thread** object using the following constructor −

Thread(Runnable threadObj, String threadName);

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

void start();

Example

Here is an example that creates a new thread and starts running it −

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

```
public class TestThread {

  public static void main(String args[]) {
    RunnableDemo R1 = new RunnableDemo( "Thread-1");
    R1.start();

    RunnableDemo R2 = new RunnableDemo( "Thread-2");
    R2.start();
  }
}
```

This will produce the following result −

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run( )** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method −

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

```
void start( );
```

Example

Here is the preceding program rewritten to extend the Thread −

```
class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;

  ThreadDemo( String name) {
    threadName = name;
```

```
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}

public class TestThread {

   public static void main(String args[]) {
      ThreadDemo T1 = new ThreadDemo( "Thread-1");
      T1.start();

      ThreadDemo T2 = new ThreadDemo( "Thread-2");
      T2.start();
   }
}
```

This will produce the following result −

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
```

```
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread Methods

Following is the list of important methods available in the Thread class.

Sr.No.	Method & Description
1	**public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

Sr.No.	Method & Description
1	**public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	**public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds.

3	**public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable** −

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable

public class DisplayMessage implements Runnable {
   private String message;

   public DisplayMessage(String message) {
      this.message = message;
   }

   public void run() {
      while(true) {
         System.out.println(message);
      }
   }
}
```

Following is another class which extends the Thread class −

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread

public class GuessANumber extends Thread {
   private int number;
   public GuessANumber(int number) {
      this.number = number;
   }

   public void run() {
      int counter = 0;
      int guess = 0;
      do {
         guess = (int) (Math.random() * 100 + 1);
         System.out.println(this.getName() + " guesses " + guess);
         counter++;
      } while(guess != number);
      System.out.println("** Correct!" + this.getName() + "in" + counter + "guesses.**");
   }
}
```

Following is the main program, which makes use of the above-defined classes −

```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo {

  public static void main(String [] args) {
    Runnable hello = new DisplayMessage("Hello");
    Thread thread1 = new Thread(hello);
    thread1.setDaemon(true);
    thread1.setName("hello");
    System.out.println("Starting hello thread...");
    thread1.start();

    Runnable bye = new DisplayMessage("Goodbye");
    Thread thread2 = new Thread(bye);
    thread2.setPriority(Thread.MIN_PRIORITY);
    thread2.setDaemon(true);
    System.out.println("Starting goodbye thread...");
    thread2.start();

    System.out.println("Starting thread3...");
    Thread thread3 = new GuessANumber(27);
    thread3.start();
    try {
      thread3.join();
    } catch (InterruptedException e) {
      System.out.println("Thread interrupted.");
    }
    System.out.println("Starting thread4...");
    Thread thread4 = new GuessANumber(75);

    thread4.start();
    System.out.println("main() is ending...");
  }
}
```

This will produce the following result. You can try this example again and again and you will get a different result every time.

Output

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
```

```
Goodbye
Goodbye
Goodbye
.......
```

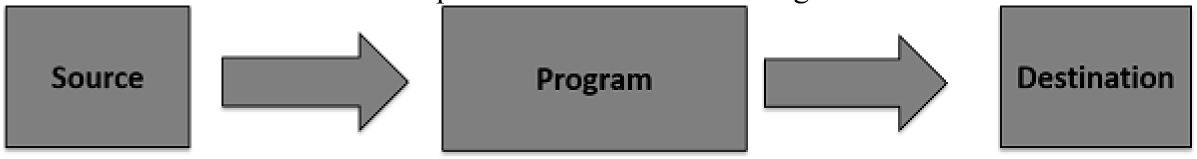
Input/output Files:

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```java
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

Example

```java
import java.io.*;
public class ReadConsole {

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

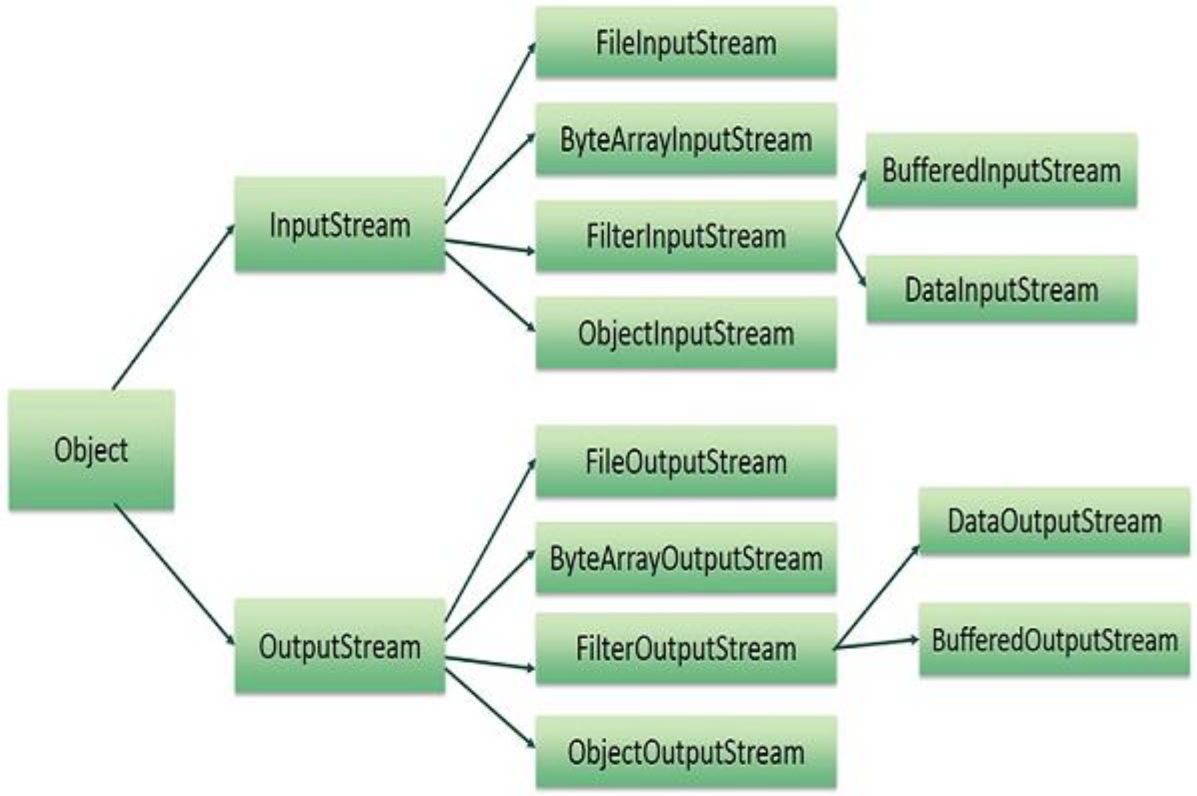
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
```

```
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.
Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.
Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links −

- ByteArrayInputStream
- DataInputStream

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.

2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links −

- ByteArrayOutputStream
- DataOutputStream

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;
public class fileStreamTest {

   public static void main(String args[]) {

      try {
         byte bWrite [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("test.txt");
         for(int x = 0; x < bWrite.length ; x++) {
            os.write( bWrite[x] );   // writes the bytes
         }
         os.close();

         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      } catch (IOException e) {
         System.out.print("Exception");
      }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class

- FileReader Class
- FileWriter Class

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

Example

```
import java.io.File;
public class CreateDir {

  public static void main(String args[]) {
    String dirname = "/tmp/user/java/bin";
    File d = new File(dirname);

    // Create directory now.
    d.mkdirs();
  }
}
```

Compile and execute the above code to create "/tmp/user/java/bin".

Note − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

Example

```
import java.io.File;
public class ReadDir {

  public static void main(String[] args) {
    File file = null;
    String[] paths;

    try {
      // create new file object
      file = new File("/tmp");

      // array of files and directory
```

```
         paths = file.list();

         // for each name in the path array
         for(String path:paths) {
            // prints filename and directory name
            System.out.println(path);
         }
      } catch (Exception e) {
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

This will produce the following result based on the directories and files available in your **/tmp** directory −

Output

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

<u>Utility Classes:</u>

Introduction

The Collections utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection,

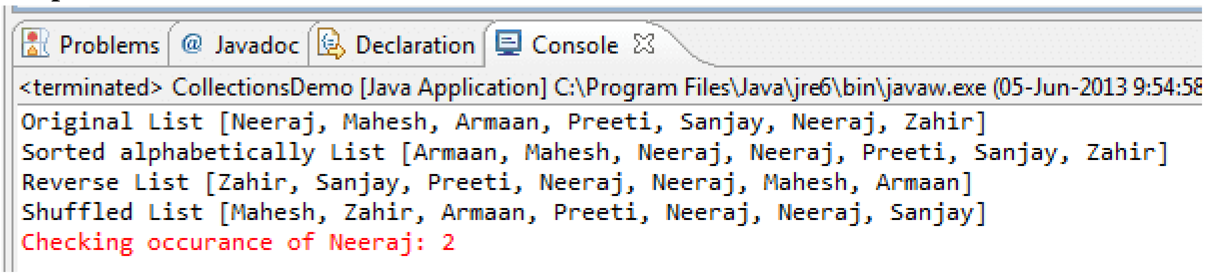
Some useful method in Collections class:

Let's take the example of List sorting using Collection class. We can sort any Collection using “Collections” utility class. i.e.; ArrayList of Strings can be sorted alphabetically using this utility class. ArrayList class itself is not providing any methods to sort. We use Collections class static methods to do this. Below program shows use of reverse(), shuffle(), frequency() methods as well.

Java Code:

```
package utility;

import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class CollectionsDemo {

          public static void main(String[] args) {
                    List<String>student<String>List = new ArrayList();
                    studentList.add("Neeraj");
                    studentList.add("Mahesh");
                    studentList.add("Armaan");
                    studentList.add("Preeti");
                    studentList.add("Sanjay");
                    studentList.add("Neeraj");
                    studentList.add("Zahir");

                    System.out.println("Original List " + studentList);

                    Collections.sort(studentList);
                    System.out.println("Sorted alphabetically List " + studentList);

                    Collections.reverse(studentList);
                    System.out.println("Reverse List " + studentList);
                    Collections.shuffle(studentList);
                    System.out.println("Shuffled List " + studentList);
                    System.out.println("Checking occurance of Neeraj: "
                                        + Collections.frequency(studentList, "Neeraj"));
          }
}
```

Output:



```
<terminated> CollectionsDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (05-Jun-2013 9:54:58
Original List [Neeraj, Mahesh, Armaan, Preeti, Sanjay, Neeraj, Zahir]
Sorted alphabetically List [Armaan, Mahesh, Neeraj, Neeraj, Preeti, Sanjay, Zahir]
Reverse List [Zahir, Sanjay, Preeti, Neeraj, Neeraj, Mahesh, Armaan]
Shuffled List [Mahesh, Zahir, Armaan, Preeti, Neeraj, Neeraj, Sanjay]
Checking occurance of Neeraj: 2
```

String Handling:

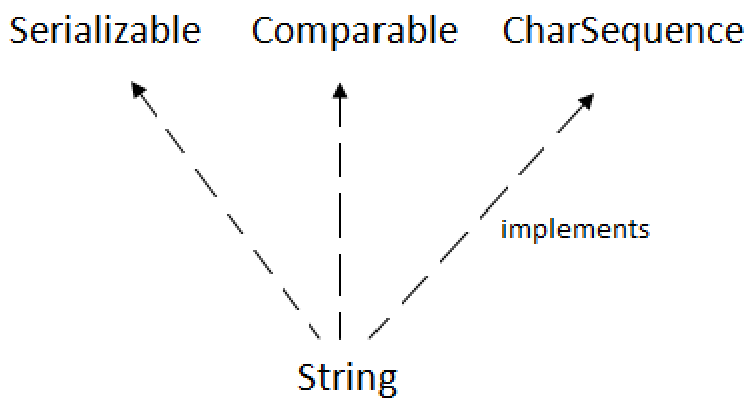
In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

1. `char[] ch={'j','a','v','a','t','p','o','i','n','t'};`
2. `String s=new String(ch);`

is same as:

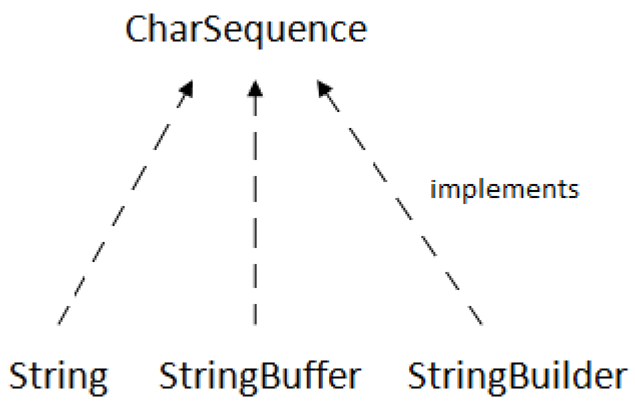
1. `String s="javatpoint";`

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters.
String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.
We will discuss immutable string later. Let's first understand what is String in Java and how to create the String object.

Java Utility Classes continue:

Method Signature	Description
Collections.sort(List myList)	Sort the myList (implementation of any List interface) provided an argument in natural ordering.
Collections.sort(List, comparator c)	Sort the myList(implementation of any List interface) as per comparator c ordering (c class should implement comparator interface)
Collections.shuffle(List myList)	Puts the elements of myList ((implementation of any List interface)in random order
Collections.reverse(List myList)	Reverses the elements of myList ((implementation of any List interface)
Collections.binarySearch(List mlist, T key)	Searches the mlist (implementation of any List interface) for the specified object using the binary search algorithm.
Collections.copy(List dest, List src)	Copy the source List into dest List.
Collections.frequency(Collection c, Object o)	Returns the number of elements in the specified collection class c (which implements Collection interface can be List, Set or Queue) equal to the specified object
Collections.synchronizedCollection(Collection c)	Returns a synchronized (thread-safe) collection backed by the specified collection.

Unit-III
JDBC

JDBC Overview – JDBC implementation – Connection class – Statements - Catching Database Results, handling database Queries. Networking– InetAddress class – URL class- TCP sockets – UDP sockets, Java Beans –RMI.

JDBC Overview:

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Why to Learn JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Applications of JDBC

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas −

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

Audience

This tutorial is designed for Java programmers who would like to understand the JDBC framework in detail along with its architecture and actual usage.

JDBC Implementation:

What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects −

- Core JAVA Programming
- SQL or MySQL Database

JDBC Architecture

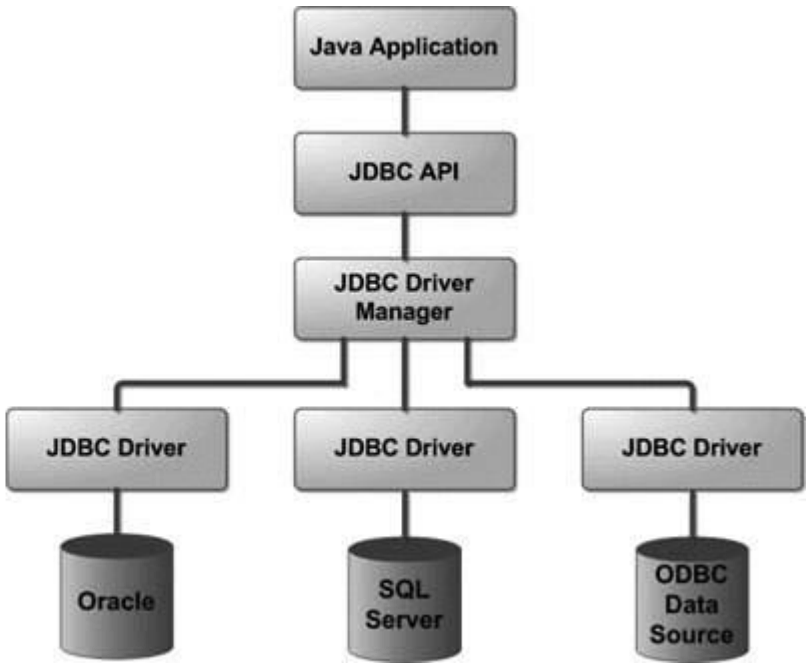
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas −

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.

- Annotations.

<u>Connection class:</u>

Create a DSN-less Database Connection
The easiest way to connect to a database is to use a DSN-less connection. A DSN-less connection can be used against any Microsoft Access database on your web site.
If you have a database called "northwind.mdb" located in a web directory like "c:/webdata/", you can connect to the database with the following ASP code:

```
<%
set conn=Server.CreateObject("ADODB.Connection")
conn.Provider="Microsoft.Jet.OLEDB.4.0"
conn.Open "c:/webdata/northwind.mdb"
%>
```

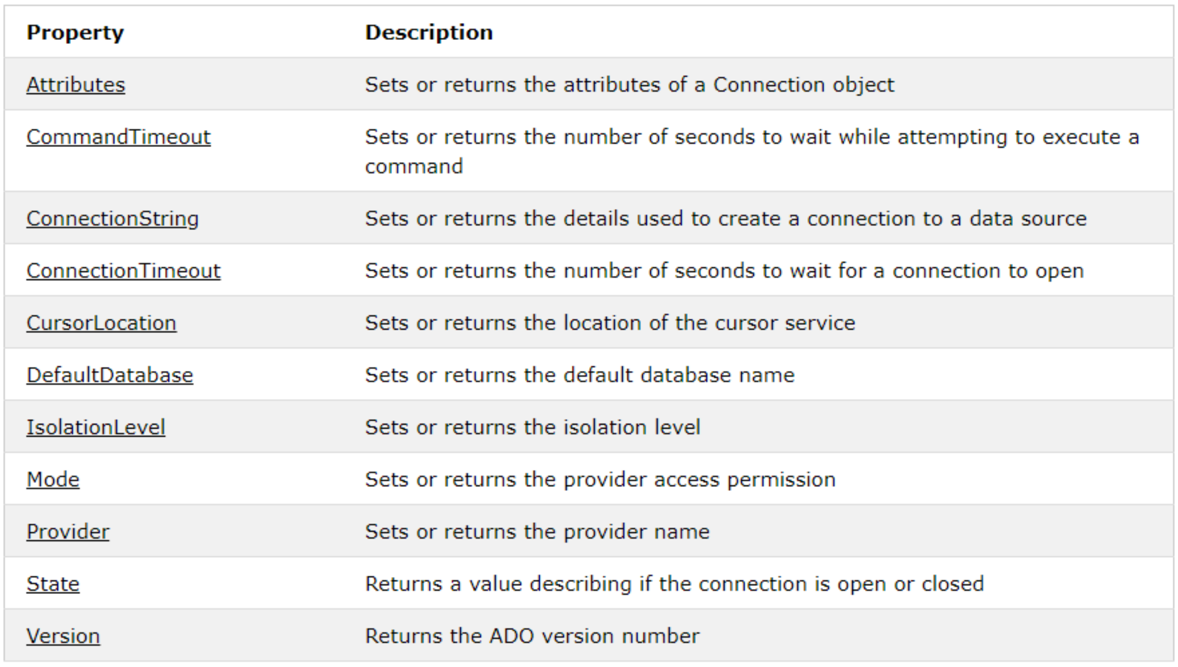
Note, from the example above, that you have to specify the Microsoft Access database driver (Provider) and the physical path to the database on your computer.
Create an ODBC Database Connection
If you have an ODBC database called "northwind" you can connect to the database with the following ASP code:

```
<%
set conn=Server.CreateObject("ADODB.Connection")
conn.Open "northwind"
%>
```

With an ODBC connection, you can connect to any database, on any computer in your network, as long as an ODBC connection is available.
An ODBC Connection to an MS Access Database
Here is how to create a connection to a MS Access Database:

1. Open the **ODBC** icon in your Control Panel.
2. Choose the **System DSN** tab.
3. Click on **Add** in the System DSN tab.
4. **Select** the Microsoft Access Driver. Click **Finish.**
5. In the next screen, click **Select** to locate the database.
6. Give the database a **D**ata **S**ource **N**ame (DSN).
7. Click **OK**.

Note that this configuration has to be done on the computer where your web site is located. If you are running Personal Web Server (PWS) or Internet Information Server (IIS) on your own computer, the instructions above will work, but if your web site is located on a remote server, you have to have physical access to that server, or ask your web host to do this for you.
The ADO Connection Object
The ADO Connection object is used to create an open connection to a data source. Through this connection, you can access and manipulate a database.



Property	Description
Attributes	Sets or returns the attributes of a Connection object
CommandTimeout	Sets or returns the number of seconds to wait while attempting to execute a command
ConnectionString	Sets or returns the details used to create a connection to a data source
ConnectionTimeout	Sets or returns the number of seconds to wait for a connection to open
CursorLocation	Sets or returns the location of the cursor service
DefaultDatabase	Sets or returns the default database name
IsolationLevel	Sets or returns the isolation level
Mode	Sets or returns the provider access permission
Provider	Sets or returns the provider name
State	Returns a value describing if the connection is open or closed
Version	Returns the ADO version number

Statements:

Once a connection is obtained we can interact with the database. The JDBC *Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.
They also define methods that help bridge data type differences between Java and SQL data types used in a database.
The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
```

```
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   stmt.close();
}
```

For a better understanding, we suggest you to study the Statement - Example tutorial.

The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
```

```
finally {
   . . .
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

For a better understanding, let us study Prepare - Example Code.

The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure −

```
CREATE OR REPLACE PROCEDURE getEmpName
   (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END;
```

NOTE: Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database −

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
  (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
  SELECT first INTO EMP_FIRST
  FROM Employees
  WHERE ID = EMP_ID;
END $$

DELIMITER ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.
Here are the definitions of each −

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure −

```
CallableStatement cstmt = null;
try {
  String SQL = "{call getEmpName (?, ?)}";
  cstmt = conn.prepareCall (SQL);
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  . . .
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders.
Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.
If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.
Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.
A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close();
}
```

Catching database Results:

This chapter provides an example of how to create a simple JDBC application. This will show you how to open a database connection, execute a SQL query, and display the results.
All the steps mentioned in this template example, would be explained in subsequent chapters of this tutorial.

Creating JDBC Application

There are following six steps involved in building a JDBC application −

- **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Sample Code

This sample example can serve as a **template** when you need to create your own JDBC application in the future.

This sample code has been written based on the environment and database setup done in the previous chapter.

Copy and paste the following example in FirstExample.java, compile and run as follows −

```
//STEP 1. Import required packages
import java.sql.*;

public class FirstExample {
  // JDBC driver name and database URL
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DB_URL = "jdbc:mysql://localhost/EMP";

  //  Database credentials
  static final String USER = "username";
  static final String PASS = "password";

  public static void main(String[] args) {
  Connection conn = null;
  Statement stmt = null;
  try{
    //STEP 2: Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");

    //STEP 3: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL,USER,PASS);

    //STEP 4: Execute a query
    System.out.println("Creating statement...");
    stmt = conn.createStatement();
    String sql;
    sql = "SELECT id, first, last, age FROM Employees";
    ResultSet rs = stmt.executeQuery(sql);

    //STEP 5: Extract data from result set
    while(rs.next()){
      //Retrieve by column name
      int id  = rs.getInt("id");
      int age = rs.getInt("age");
      String first = rs.getString("first");
      String last = rs.getString("last");

      //Display values
      System.out.print("ID: " + id);
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);
    }
    //STEP 6: Clean-up environment
    rs.close();
    stmt.close();
```

```
      conn.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

Now let us compile the above example as follows −

```
C:\>javac FirstExample.java
C:\>
```

When you run **FirstExample**, it produces the following result −

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

Handling Database Queries:

The statement interface provides methods that are used to execute static SQL statements. The SQL statements are queries, insertions, updates and deletions, etc.
Some of the methods of the Statement interface are as follows.
• **void close():** This method closes database connections associated with Statement's object and releases JDBC resources.
• **boolean execute (String sql):** This method is used to execute an SQL statement that might produce multiple result sets or multiple counts. It returns true if multiple result sets are produced and returns false if multiple counts are generated.

• **int executeUpdate(String sql):** This method is used to execute an SQL statement that gives information about number of affected rows. For example, if we execute the statements like INSERT, DELETE, the result will only be a number of rows affected.
• **ResultSet executeQuery(String sql):** This method executes a query and returns a Resu1ts et object. A result set isjust like a table containing the resultant data.
• **ResultSet getResultSet () :** This method returns the first result set or multiple count generated on the execution of execute (String sql) method. If there is no result set, it returns null value.
• **int getUpdateCount**():After the execution of the execute (String sql) method updates counts may be generated. This method returns the first update count and clears it. More update counts can be retrieved by getMoreResults ().Note that this method returns value -1 if there is no update count or when only result sets are generated on execution of execute (String sql) method.
• **boolean getMoreResults () :** This method is used to retrieve the next result set in multiple result set or to the next count in multiple update count generated on execution of execute (String sql) method. It returns a true value if multiple sets are available and returns a false value if multiple update counts are available.

Networking:

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Why to Learn JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Applications of JDBC

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.
JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.
The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.
The new features in these packages include changes in the following areas −

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

InetAddress Class:

Java InetAddress class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name *for example* www.javatpoint.com, www.google.com, www.facebook.com, etc.
An IP address is represented by 32-bit or 128-bit unsigned number. An instance of InetAddress represents the IP address with its corresponding host name. There are two types of address types: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.
Moreover, InetAddress has a cache mechanism to store successful and unsuccessful host name resolutions.

Commonly used methods of InetAddress class

Method	Description
public static InetAddress getByName(String host) throws UnknownHostException	it returns the instance of InetAddress containing LocalHost IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	it returns the instance of InetAdddress containing local host name and address.
public String getHostName()	it returns the host name of the IP address.
public String getHostAddress()	it returns the IP address in string format.

Example of Java InetAddress class

Let's see a simple example of InetAddress class to get ip address of www.javatpoint.com website.

```
import java.io.*;
import java.net.*;
public class InetDemo{
public static void main(String[] args){
try{
InetAddress ip=InetAddress.getByName("www.javatpoint.com");

System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}catch(Exception e){System.out.println(e);}
}
}
```

Output:

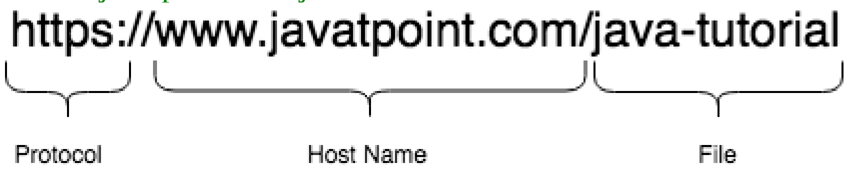
```
Host Name: www.javatpoint.com
IP Address: 206.51.231.148
```

URL Class:

Java URL

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

```
https://www.javatpoint.com/java-tutorial
```



A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.
3. **Port Number:** It is an optional attribute. If we write http//ww.javatpoint.com:80/sonoojaiswal/ , 80 is the port number. If port number is not mentioned in the URL, it returns -1.
4. **File Name or directory name:** In this case, index.jsp is the file name.

Constructors of Java URL class

URL(String spec)

Creates an instance of a URL from the String representation.

URL(String protocol, String host, int port, String file)

Creates an instance of a URL from the given protocol, host, port number, and file.

URL(String protocol, String host, int port, String file, URLStreamHandler handler)

Creates an instance of a URL from the given protocol, host, port number, file, and handler.

URL(String protocol, String host, String file)

Creates an instance of a URL from the given protocol name, host name, and file name.

URL(URL context, String spec)

Creates an instance of a URL by parsing the given spec within a specified context.

URL(URL context, String spec, URLStreamHandler handler)

Creates an instance of a URL by parsing the given spec with the specified handler within a given context.

Commonly used methods of Java URL class

The java.net.URL class provides many methods. The important methods of URL class are given below.

Method	Description
public String getProtocol()	it returns the protocol of the URL.
public String getHost()	it returns the host name of the URL.
public String getPort()	it returns the Port Number of the URL.
public String getFile()	it returns the file name of the URL.
public String getAuthority()	it returns the authority of the URL.
public String toString()	it returns the string representation of the URL.
public String getQuery()	it returns the query string of the URL.
public String getDefaultPort()	it returns the default port of the URL.
public URLConnection openConnection()	it returns the instance of URLConnection i.e. associated with this URL.
public boolean equals(Object obj)	it compares the URL with the given object.
public Object getContent()	it returns the content of the URL.
public String getRef()	it returns the anchor or reference of the URL.
public URI toURI()	it returns a URI of the URL.

Example of Java URL class

```
//URLDemo.java
import java.net.*;
public class URLDemo{
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");

System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("File Name: "+url.getFile());

}catch(Exception e){System.out.println(e);}
}
}
```

Test it Now

Output:

```
Protocol: http
Host Name: www.javatpoint.com
Port Number: -1
File Name: /java-tutorial
```

Let us see another example URL class in Java.

```java
//URLDemo.java
import java.net.*;
public class URLDemo{
public static void main(String[] args){
try{
URL url=new URL("https://www.google.com/search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8");

System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("Default Port Number: "+url.getDefaultPort());
System.out.println("Query String: "+url.getQuery());
System.out.println("Path: "+url.getPath());
System.out.println("File: "+url.getFile());

}catch(Exception e){System.out.println(e);}
}
}
```

Output:

```
Protocol: https
Host Name: www.google.com
Port Number: -1
Default Port Number: 443
Query String: q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
Path: /search
File: /search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
```

TCP Sockets:

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

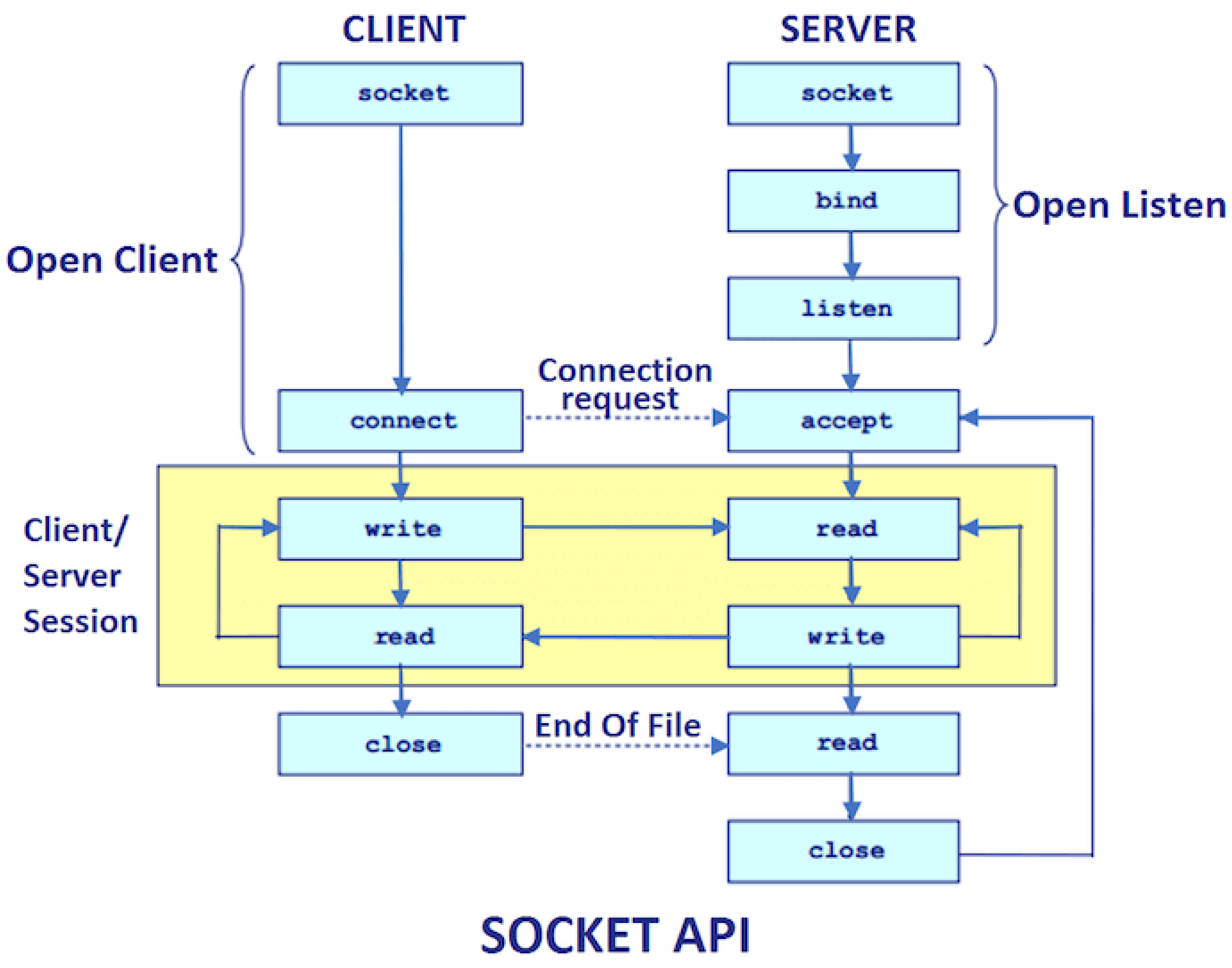
Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



Socket class
A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

ServerSocket class
The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.
Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between serve
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming
Creating Server:
To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You

may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

```java
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection and waits for the client
```

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

```java
Socket s=new Socket("localhost",6666);
```

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

File: MyServer.java

```java
import java.io.*;
import java.net.*;
public class MyServer {
public static void main(String[] args){
try{
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection
DataInputStream dis=new DataInputStream(s.getInputStream());
String  str=(String)dis.readUTF();
System.out.println("message= "+str);
ss.close();
}catch(Exception e){System.out.println(e);}
}
}
```

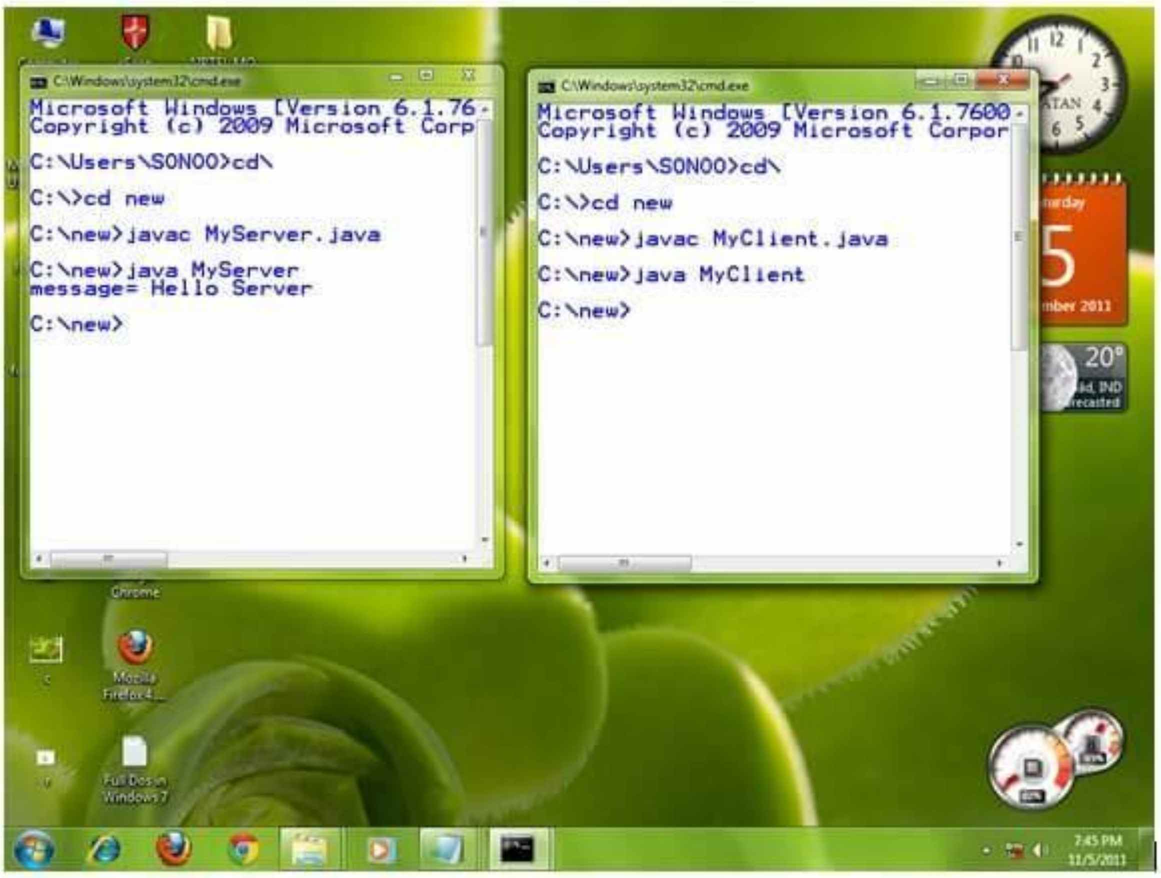
File: MyClient.java

```java
import java.io.*;
import java.net.*;
public class MyClient {
public static void main(String[] args) {
try{
Socket s=new Socket("localhost",6666);
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
dout.writeUTF("Hello Server");
dout.flush();
dout.close();
s.close();
}catch(Exception e){System.out.println(e);}
}
}
```

download this example

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



```
Microsoft Windows [Version 6.1.76
Copyright (c) 2009 Microsoft Corp

C:\Users\SONOO>cd\

C:\>cd new

C:\new>javac MyServer.java

C:\new>java MyServer
message= Hello Server

C:\new>
```

```
Microsoft Windows [Version 6.1.7600
Copyright (c) 2009 Microsoft Corpor

C:\Users\SONOO>cd\

C:\>cd new

C:\new>javac MyClient.java

C:\new>java MyClient

C:\new>
```

Example of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

File: MyServer.java

```java
import java.net.*;
import java.io.*;
class MyServer{
public static void main(String args[])throws Exception{
ServerSocket ss=new ServerSocket(3333);
Socket s=ss.accept();
DataInputStream din=new DataInputStream(s.getInputStream());
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

String str="",str2="";
while(!str.equals("stop")){
str=din.readUTF();
System.out.println("client says: "+str);
str2=br.readLine();
dout.writeUTF(str2);
dout.flush();
}
din.close();
s.close();
ss.close();
}}
```

File: MyClient.java

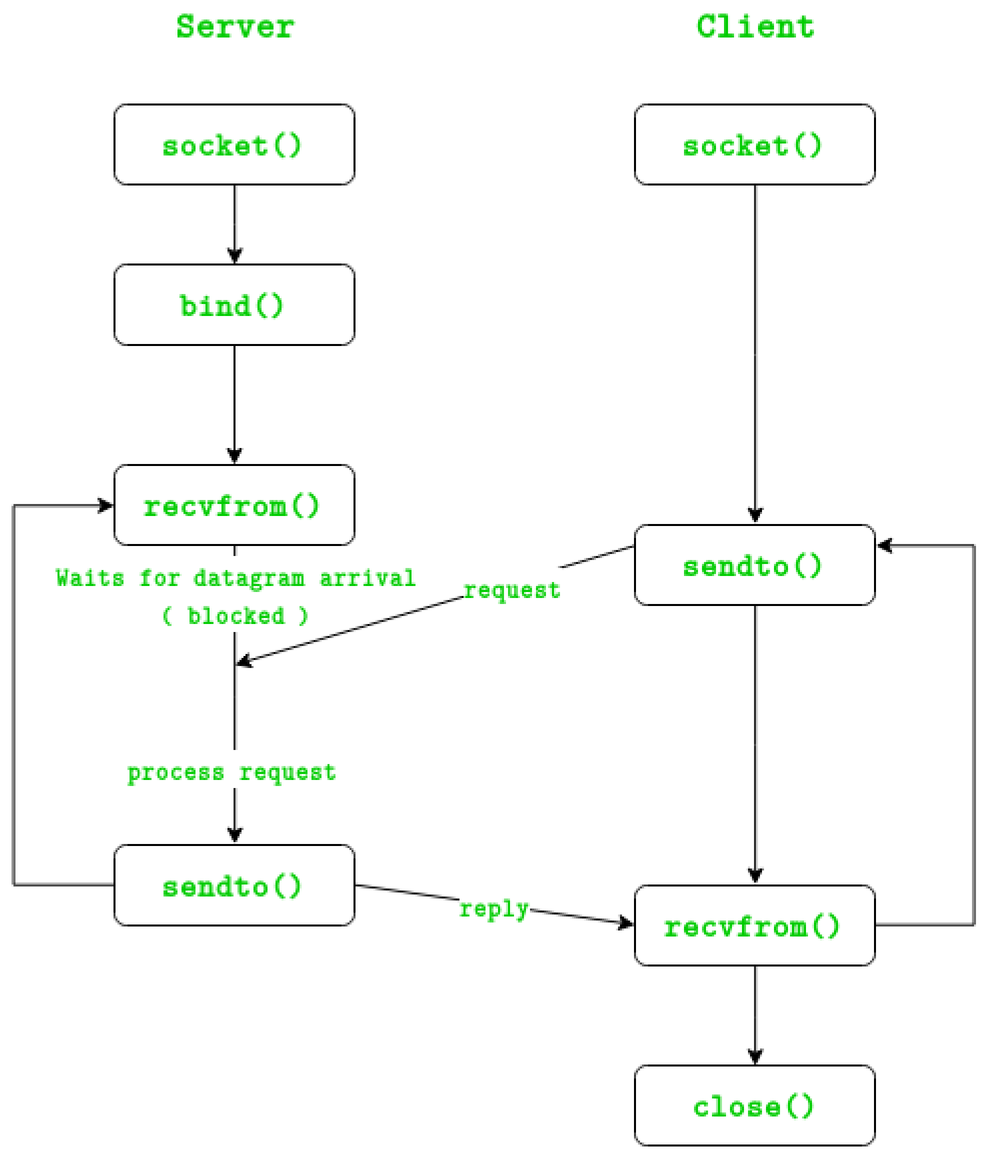
```
import java.net.*;
import java.io.*;
class MyClient{
public static void main(String args[])throws Exception{
Socket s=new Socket("localhost",3333);
DataInputStream din=new DataInputStream(s.getInputStream());
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

String str="",str2="";
while(!str.equals("stop")){
str=br.readLine();
dout.writeUTF(str);
dout.flush();
str2=din.readUTF();
System.out.println("Server says: "+str2);
}

dout.close();
s.close();
}}
```

UDP Sockets:

UDP Server-Client implementation in C

There are two major transport layer protocols to communicate between hosts : **TCP** and **UDP**. Creating TCP Server/Client was discussed in a previous post.
Prerequisite : Creating TCP Server/Client
Theory
In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data

to the correct client.



The entire process can be broken down into following steps :

UDP Server :

1. Create UDP socket.
2. Bind the socket to server address.
3. Wait until datagram packet arrives from client.
4. Process the datagram packet and send a reply to client.
5. Go back to Step 3.

UDP Client :

1. Create UDP socket.
2. Send message to server.
3. Wait until response from server is recieved.
4. Process reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

Necessary Functions :

```
int socket(int domain, int type, int protocol)
Creates an unbound socket in the specified domain.
Returns socket file descriptor.
```

Arguments :
domain – Specifies the communication
domain (AF_INET for IPv4/ AF_INET6 for IPv6)
type – Type of socket to be created
(SOCK_STREAM for TCP / SOCK_DGRAM for UDP)
protocol – Protocol to be used by socket.
0 means use default protocol for the address family.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
Assigns address to the unbound socket.
```

Arguments :
sockfd – File descriptor of socket to be binded
addr – Structure in which address to be binded to is specified
addrlen – Size of *addr* structure

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
            const struct sockaddr *dest_addr, socklen_t addrlen)
Send a message on the socket
```

Arguments :
sockfd – File descriptor of socket
buf – Application buffer containing the data to be sent
len – Size of *buf* application buffer
flags – Bitwise OR of flags to modify socket behaviour
dest_addr – Structure containing address of destination
addrlen – Size of *dest_addr* structure

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
             struct sockaddr *src_addr, socklen_t *addrlen)
Receive a message from the socket.
```

Arguments :
sockfd – File descriptor of socket
buf – Application buffer in which to receive data
len – Size of *buf* application buffer
flags – Bitwise OR of flags to modify socket behaviour
src_addr – Structure containing source address is returned
addrlen – Variable in which size of *src_addr* structure is returned

```
int close(int fd)
Close a file descriptor
```

Arguments :
fd – File descriptor
In the below code, exchange of one hello message between server and client is shown to demonstrate the model.

- UDPClient.c

filter_none

edit
play_arrow
brightness_4

```
// Client side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024

// Driver code
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in     servaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                MSG_WAITALL, (struct sockaddr *) &servaddr,
                &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd);
```

```
   return 0;
}
```

Output :

```
$ ./server
Client : Hello from client
Hello message sent.
$ ./client
Hello message sent.
Server : Hello from server
```

Java Beans:

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.
Following are the unique characteristics that distinguish a JavaBean from other Java classes −

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "**getter**" and "**setter**" methods for the properties.

JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.
A JavaBean property may be **read, write, read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class −

S.No.	Method & Description
1	get**PropertyName**() For example, if property name is *firstName*, your method name would be **getFirstName()** to read that property. This method is called accessor.
2	set**PropertyName**() For example, if property name is *firstName*, your method name would be **setFirstName()** to write that property. This method is called mutator.

A read-only attribute will have only a **getPropertyName()** method, and a write-only attribute will have only a **setPropertyName()** method.

JavaBeans Example

Consider a student class with few properties −

```
package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable {
   private String firstName = null;
   private String lastName = null;
   private int age = 0;

   public StudentsBean() {
   }
   public String getFirstName(){
      return firstName;
```

```
  }
  public String getLastName(){
    return lastName;
  }
  public int getAge(){
    return age;
  }
  public void setFirstName(String firstName){
    this.firstName = firstName;
  }
  public void setLastName(String lastName){
    this.lastName = lastName;
  }
  public void setAge(Integer age){
    this.age = age;
  }
}
```

Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action −

```
<html>
  <head>
    <title>useBean Example</title>
  </head>

  <body>
    <jsp:useBean id = "date" class = "java.util.Date" />
    <p>The date/time is <%= date %>
  </body>
</html>
```

You will receive the following result − −

The date/time is Thu Sep 30 11:18:11 GST 2010

Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax −

```
<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
  <jsp:setProperty name = "bean's id" property = "property name"
    value = "value"/>
  <jsp:getProperty name = "bean's id" property = "property name"/>
  ...........
</jsp:useBean>
```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.
Following example shows how to access the data using the above syntax −

```
<html>
  <head>
    <title>get and set properties Example</title>
  </head>

  <body>
    <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">
      <jsp:setProperty name = "students" property = "firstName" value = "Zara"/>
      <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>
      <jsp:setProperty name = "students" property = "age" value = "10"/>
    </jsp:useBean>

    <p>Student First Name:
      <jsp:getProperty name = "students" property = "firstName"/>
    </p>

    <p>Student Last Name:
      <jsp:getProperty name = "students" property = "lastName"/>
    </p>

    <p>Student Age:
      <jsp:getProperty name = "students" property = "age"/>
    </p>

  </body>
</html>
```

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed −

```
Student First Name: Zara

Student Last Name: Ali

Student Age: 10
```

<u>RMI:</u>

In the previous chapter, we created a sample RMI application where a client invokes a method which displays a GUI window (JavaFX).
In this chapter, we will take an example to see how a client program can retrieve the records of a table in MySQL database residing on the server.
Assume we have a table named **student_data** in the database **details** as shown below.

```
+----+--------+--------+------------+---------------------+
| ID | NAME   | BRANCH | PERCENTAGE | EMAIL               |
+----+--------+--------+------------+---------------------+
|  1 | Ram    | IT     |         85 | ram123@gmail.com    |
|  2 | Rahim  | EEE    |         95 | rahim123@gmail.com  |
|  3 | Robert | ECE    |         90 | robert123@gmail.com |
+----+--------+--------+------------+---------------------+
```

Assume the name of the user is **myuser** and its password is **password**.

Creating a Student Class

Create a **Student** class with **setter** and **getter** methods as shown below.

```
public class Student implements java.io.Serializable {
   private int id, percent;
   private String name, branch, email;

   public int getId() {
      return id;
   }
   public String getName() {
      return name;
   }
   public String getBranch() {
      return branch;
   }
   public int getPercent() {
      return percent;
   }
   public String getEmail() {
      return email;
   }
   public void setID(int id) {
      this.id = id;
   }
   public void setName(String name) {
      this.name = name;
   }
   public void setBranch(String branch) {
      this.branch = branch;
   }
   public void setPercent(int percent) {
      this.percent = percent;
   }
   public void setEmail(String email) {
      this.email = email;
   }
}
```

Defining the Remote Interface

Define the remote interface. Here, we are defining a remote interface named **Hello** with a method named **getStudents ()** in it. This method returns a list which contains the object of the class **Student**.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;

// Creating Remote interface for our application
```

```
public interface Hello extends Remote {
  public List<Student> getStudents() throws Exception;
}
```

Developing the Implementation Class

Create a class and implement the above created **interface.**

Here we are implementing the **getStudents()** method of the **Remote interface**. When you invoke this method, it retrieves the records of a table named **student_data**. Sets these values to the Student class using its setter methods, adds it to a list object and returns that list.

```
import java.sql.*;
import java.util.*;

// Implementing the remote interface
public class ImplExample implements Hello {

  // Implementing the interface method
  public List<Student> getStudents() throws Exception {
    List<Student> list = new ArrayList<Student>();

    // JDBC driver name and database URL
    String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    String DB_URL = "jdbc:mysql://localhost:3306/details";

    // Database credentials
    String USER = "myuser";
    String PASS = "password";

    Connection conn = null;
    Statement stmt = null;

    //Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");

    //Open a connection
    System.out.println("Connecting to a selected database...");
    conn = DriverManager.getConnection(DB_URL, USER, PASS);
    System.out.println("Connected database successfully...");

    //Execute a query
    System.out.println("Creating statement...");

    stmt = conn.createStatement();
    String sql = "SELECT * FROM student_data";
    ResultSet rs = stmt.executeQuery(sql);

    //Extract data from result set
    while(rs.next()) {
      // Retrieve by column name
      int id  = rs.getInt("id");

      String name = rs.getString("name");
```

```
        String branch = rs.getString("branch");

        int percent = rs.getInt("percentage");
        String email = rs.getString("email");

        // Setting the values
        Student student = new Student();
        student.setID(id);
        student.setName(name);
        student.setBranch(branch);
        student.setPercent(percent);
        student.setEmail(email);
        list.add(student);
      }
      rs.close();
      return list;
    }
}
```

Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMI registry**.

Following is the server program of this application. Here, we will extend the above created class, create a remote object and register it to the RMI registry with the bind name ***hello***.

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server extends ImplExample {
  public Server() {}
  public static void main(String args[]) {
    try {
      // Instantiating the implementation class
      ImplExample obj = new ImplExample();

      // Exporting the object of implementation class (
        here we are exporting the remote object to the stub)
      Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

      // Binding the remote object (stub) in the registry
      Registry registry = LocateRegistry.getRegistry();

      registry.bind("Hello", stub);
      System.err.println("Server ready");
    } catch (Exception e) {
      System.err.println("Server exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

Client Program

Following is the client program of this application. Here, we are fetching the remote object and invoking the method named **getStudents()**. It retrieves the records of the table from the list object and displays them.

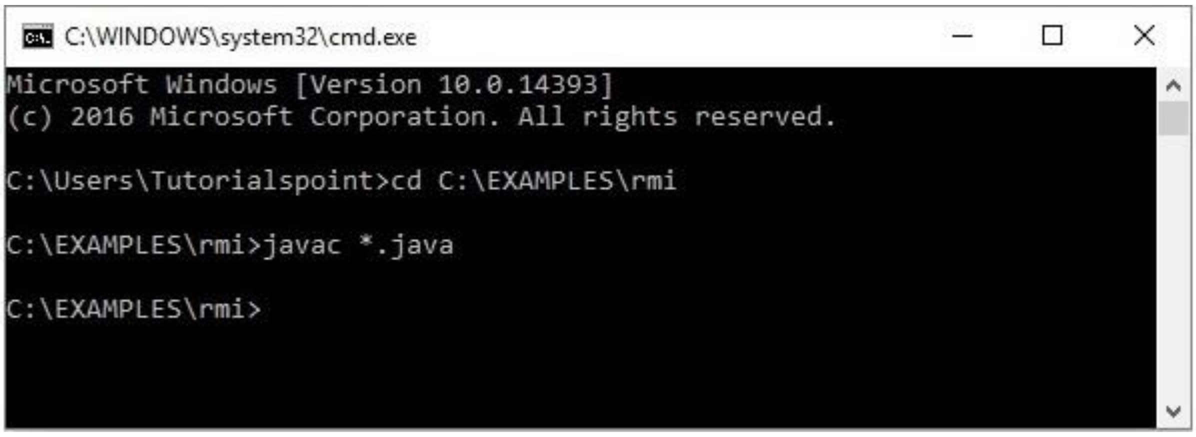
```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.*;

public class Client {
  private Client() {}
  public static void main(String[] args)throws Exception {
    try {
      // Getting the registry
      Registry registry = LocateRegistry.getRegistry(null);

      // Looking up the registry for the remote object
      Hello stub = (Hello) registry.lookup("Hello");

      // Calling the remote method using the obtained object
      List<Student> list = (List)stub.getStudents();
      for (Student s:list)v {

        // System.out.println("bc "+s.getBranch());
        System.out.println("ID: " + s.getId());
        System.out.println("name: " + s.getName());
        System.out.println("branch: " + s.getBranch());
        System.out.println("percent: " + s.getPercent());
        System.out.println("email: " + s.getEmail());
      }
    // System.out.println(list);
    } catch (Exception e) {
      System.err.println("Client exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

Steps to Run the Example

Following are the steps to run our RMI Example.

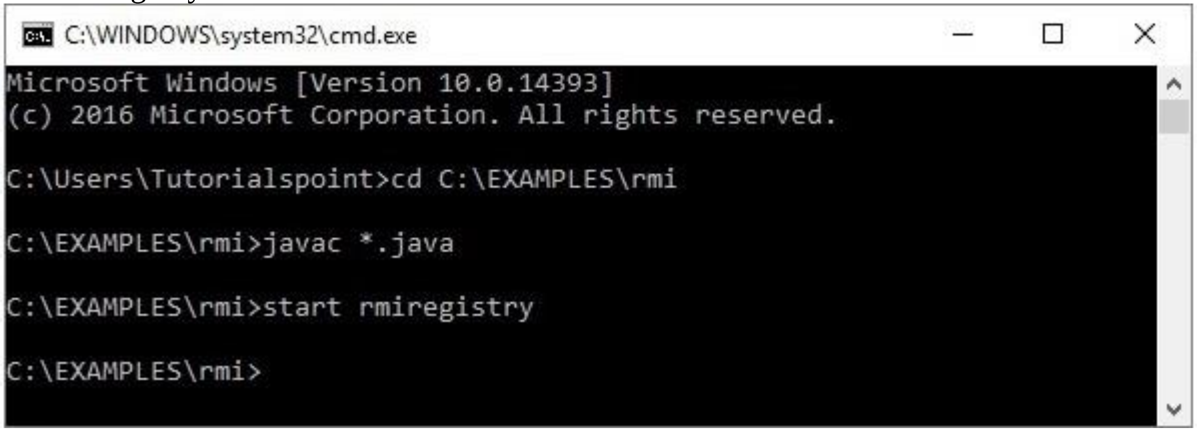
Step 1 − Open the folder where you have stored all the programs and compile all the Java files as shown below.

```
Javac *.java
```



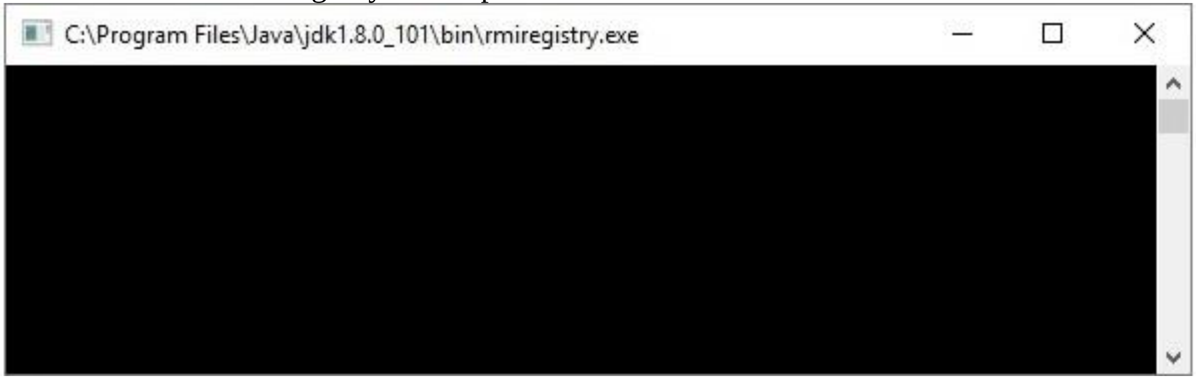
```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>
```

Step 2 − Start the **rmi** registry using the following command.
start rmiregistry

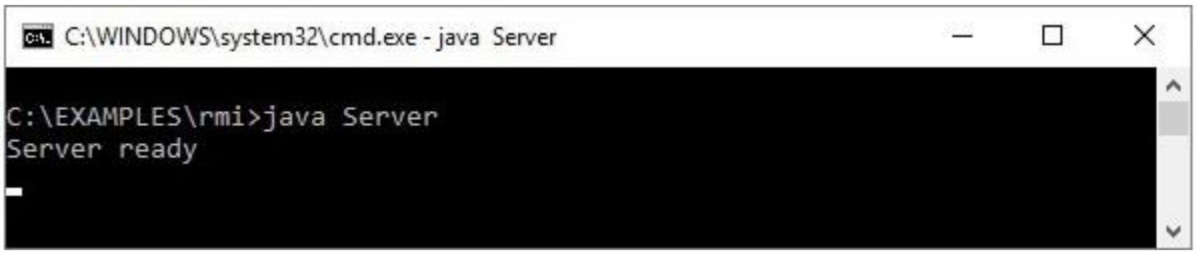


```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>
```

This will start an **rmi** registry on a separate window as shown below.

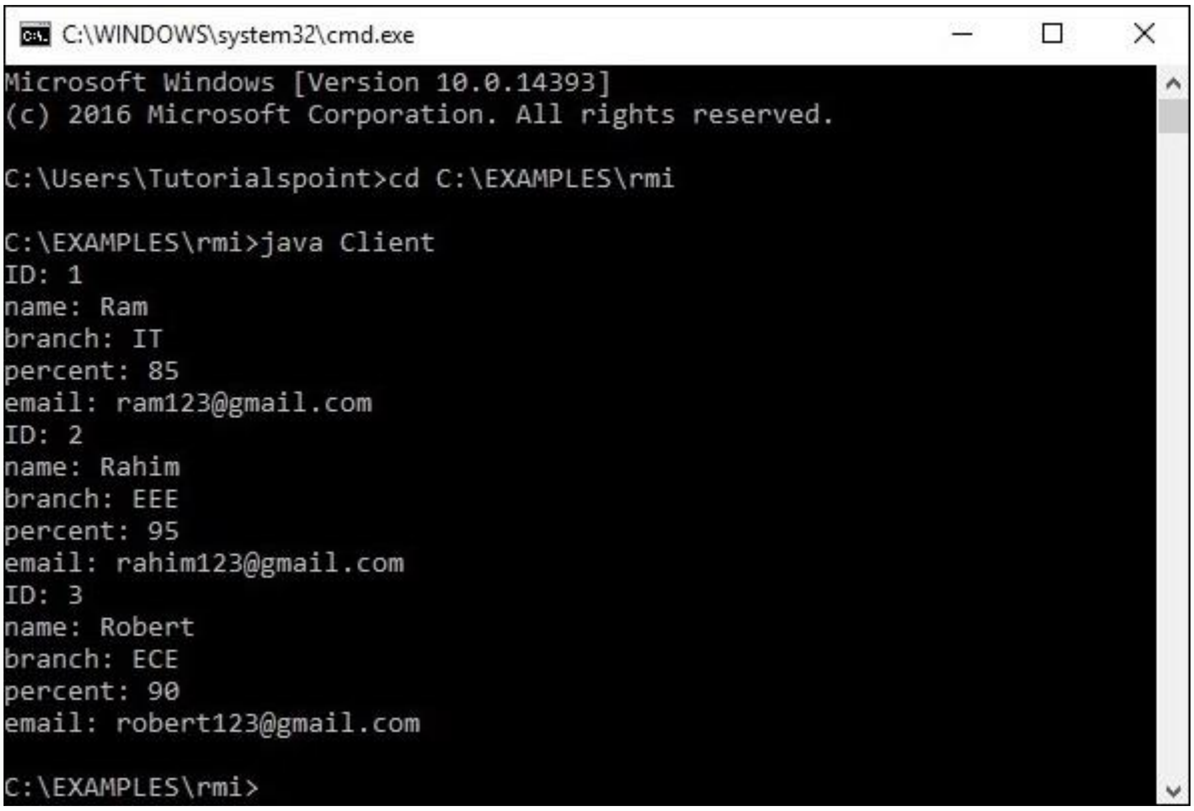


Step 3 − Run the server class file as shown below.
Java Server



```
C:\EXAMPLES\rmi>java Server
Server ready
```

Step 4 − Run the client class file as shown below.
java Client



```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>java Client
ID: 1
name: Ram
branch: IT
percent: 85
email: ram123@gmail.com
ID: 2
name: Rahim
branch: EEE
percent: 95
email: rahim123@gmail.com
ID: 3
name: Robert
branch: ECE
percent: 90
email: robert123@gmail.com

C:\EXAMPLES\rmi>
```

Unit-IV

APPLETS

Java applets- Life cycle of an applet – Adding images to an applet – Adding sound to an applet. Passing parameters to an applet. Event Handling. Introducing AWT: Working with Windows Graphics and Text. Using AWT Controls, Layout Managers and Menus. Servlet – life cycle of a servlet. The Servlet API, Handling HTTP Request and Response, using Cookies, Session Tracking. Introduction to JSP.

Java Applets:

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
  public void paint (Graphics g) {
    g.drawString ("Hello World", 25, 50);
  }
}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Following is an example that invokes the "Hello, World" applet −

```
<html>
  <title>The Hello, World Applet</title>
  <hr>
  <applet code = "HelloWorldApplet.class" width = "320" height = "120">
    If your browser was Java-enabled, a "Hello, World"
    message would appear here.
  </applet>
  <hr>
</html>
```

Note − You can refer to HTML Applet Tag to understand more about calling applet from HTML.
The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with an </applet> tag.
If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.
Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.
The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown −

```
<applet codebase = "https://amrood.com/applets" code = "HelloWorldApplet.class"
  width = "320" height = "120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example −

```
<applet  = "mypackage.subpackage.TestApplet.class"
  width = "320" height = "120">
```

Getting Applet Parameters

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color.
The second color and the size of each square may be specified as parameters to the applet within the document.
CheckerApplet gets its parameters in the init() method. It may also get its parameters in the paint() method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.
The applet viewer or browser calls the init() method of each applet it runs. The viewer calls init() once, immediately after loading the applet. (Applet.init() is implemented to do nothing.) Override the default implementation to insert custom initialization code.
The Applet.getParameter() method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.
The following is a skeleton of CheckerApplet.java −

```
import java.applet.*;
import java.awt.*;

public class CheckerApplet extends Applet {
  int squareSize = 50;   // initialized to default size
  public void init() {}
  private void parseSquareSize (String param) {}
  private Color parseColor (String param) {}
  public void paint (Graphics g) {}
}
```

Here are CheckerApplet's init() and private parseSquareSize() methods −

```
public void init () {
  String squareSizeParam = getParameter ("squareSize");
  parseSquareSize (squareSizeParam);

  String colorParam = getParameter ("color");
  Color fg = parseColor (colorParam);

  setBackground (Color.black);
  setForeground (fg);
}

private void parseSquareSize (String param) {
  if (param == null) return;
  try {
    squareSize = Integer.parseInt (param);
  } catch (Exception e) {
    // Let default value remain
  }
}
```

The applet calls parseSquareSize() to parse the squareSize parameter. parseSquareSize() calls the library method Integer.parseInt(), which parses a string and returns an integer. Integer.parseInt() throws an exception whenever its argument is invalid.

Therefore, parseSquareSize() catches exceptions, rather than allowing the applet to fail on bad input.

The applet calls parseColor() to parse the color parameter into a Color value. parseColor() does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet work.

Specifying Applet Parameters

The following is an example of an HTML file with a CheckerApplet embedded in it. The HTML file specifies both parameters to the applet by means of the <param> tag.

```
<html>
  <title>Checkerboard Applet</title>
  <hr>
  <applet code = "CheckerApplet.class" width = "480" height = "320">
    <param name = "color" value = "blue">
    <param name = "squaresize" value = "30">
  </applet>
  <hr>
</html>
```

Note − Parameter names are not case sensitive.

Application Conversion to Applets

It is easy to convert a graphical Java application (that is, an application that uses the AWT and that you can start with the Java program launcher) into an applet that you can embed in a web page.

Following are the specific steps for converting an application to an applet.

- Make an HTML page with the appropriate tag to load the applet code.
- Supply a subclass of the JApplet class. Make this class public. Otherwise, the applet cannot be loaded.
- Eliminate the main method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.

- Move any initialization code from the frame window constructor to the init method of the applet. You don't need to explicitly construct the applet object. The browser instantiates it for you and calls the init method.
- Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.
- Remove the call to setDefaultCloseOperation. An applet cannot be closed; it terminates when the browser exits.
- If the application calls setTitle, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML title tag.)
- Don't call setVisible(true). The applet is displayed automatically.

Event Handling

Applets inherit a group of event-handling methods from the Container class. The Container class defines several methods, such as processKeyEvent and processMouseEvent, for handling particular types of events, and then one catch-all method called processEvent.
In order to react to an event, an applet must override the appropriate event-specific method.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet implements MouseListener {
  StringBuffer strBuffer;

  public void init() {
    addMouseListener(this);
    strBuffer = new StringBuffer();
    addItem("initializing the apple ");
  }

  public void start() {
    addItem("starting the applet ");
  }

  public void stop() {
    addItem("stopping the applet ");
  }

  public void destroy() {
    addItem("unloading the applet");
  }

  void addItem(String word) {
    System.out.println(word);
    strBuffer.append(word);
    repaint();
  }

  public void paint(Graphics g) {
    // Draw a Rectangle around the applet's display area.
    g.drawRect(0, 0,
    getWidth() - 1,
    getHeight() - 1);

    // display the string inside the rectangle.
    g.drawString(strBuffer.toString(), 10, 20);
  }


  public void mouseEntered(MouseEvent event) {
  }
  public void mouseExited(MouseEvent event) {
  }
  public void mousePressed(MouseEvent event) {
  }
  public void mouseReleased(MouseEvent event) {
  }
  public void mouseClicked(MouseEvent event) {
    addItem("mouse clicked! ");
  }
}
```

Now, let us call this applet as follows −

```
<html>
   <title>Event Handling</title>
   <hr>
   <applet code = "ExampleEventHandling.class"
      width = "300" height = "300">
   </applet>
   <hr>
</html>
```

Initially, the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle, "mouse clicked" will be displayed as well.

Displaying Images

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within the applet, you use the drawImage() method found in the java.awt.Graphics class.

Following is an example illustrating all the steps to show images −

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class ImageDemo extends Applet {
   private Image image;
   private AppletContext context;

   public void init() {
      context = this.getAppletContext();
      String imageURL = this.getParameter("image");
      if(imageURL == null) {
         imageURL = "java.jpg";
      }
      try {
         URL url = new URL(this.getDocumentBase(), imageURL);
         image = context.getImage(url);
      } catch (MalformedURLException e) {
         e.printStackTrace();
         // Display in browser status bar
         context.showStatus("Could not load image!");
      }
   }

   public void paint(Graphics g) {
      context.showStatus("Displaying image");
      g.drawImage(image, 0, 0, 200, 84, null);
      g.drawString("www.javalicense.com", 35, 100);
   }
}
```

Now, let us call this applet as follows −

```
<html>
   <title>The ImageDemo applet</title>
   <hr>
   <applet code = "ImageDemo.class" width = "300" height = "200">
      <param name = "image" value = "java.jpg">
   </applet>
   <hr>
</html>
```

Playing Audio

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including −

- **public void play()** − Plays the audio clip one time, from the beginning.
- **public void loop()** − Causes the audio clip to replay continually.
- **public void stop()** − Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is an example illustrating all the steps to play an audio −

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class AudioDemo extends Applet {
   private AudioClip clip;
   private AppletContext context;

   public void init() {
      context = this.getAppletContext();
      String audioURL = this.getParameter("audio");
      if(audioURL == null) {
         audioURL = "default.au";
      }
      try {
         URL url = new URL(this.getDocumentBase(), audioURL);
         clip = context.getAudioClip(url);
      } catch (MalformedURLException e) {
         e.printStackTrace();
         context.showStatus("Could not load audio file!");
      }
   }

   public void start() {
      if(clip != null) {
         clip.loop();
      }
   }

   public void stop() {
      if(clip != null) {
         clip.stop();
      }
   }
}
```

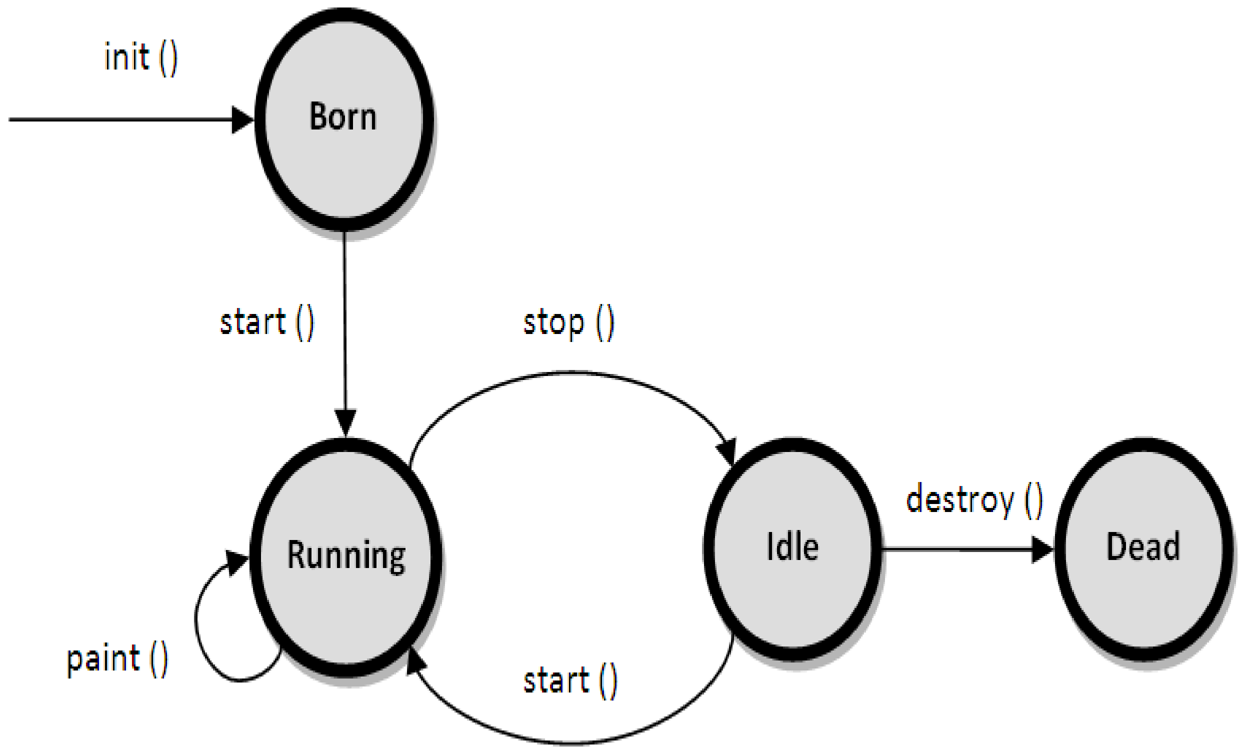
Now, let us call this applet as follows −

```
<html>
   <title>The ImageDemo applet</title>
   <hr>
   <applet code = "ImageDemo.class" width = "0" height = "0">
      <param name = "audio" value = "test.wav">
   </applet>
   <hr>
</html>
```

Life Cycle of an Applet:

In this article we will learn about applet life cycle and various life cycle methods of an applet along with example program.

The life cycle of an applet is as shown in the figure below:



As shown in the above diagram, the life cycle of an applet starts with *init()* method and ends with *destroy()* method. Other life cycle methods are *start()*, *stop()* and *paint()*. The methods to execute only once in the applet life cycle are *init()* and *destroy()*. Other methods execute multiple times.

Below is the description of each applet life cycle method:

init(): The init() method is the first method to execute when the applet is executed. Variable declaration and initialization operations are performed in this method.
start(): The start() method contains the actual code of the applet that should run. The start() method executes immediately after the *init()* method. It also executes whenever the applet is restored, maximized or moving from one tab to another tab in the browser.

stop(): The stop() method stops the execution of the applet. The stop() method executes when the applet is minimized or when moving from one tab to another in the browser.

destroy(): The destroy() method executes when the applet window is closed or when the tab containing the webpage is closed. *stop()* method executes just before when destroy() method is invoked. The destroy() method removes the applet object from memory.

paint(): The paint() method is used to redraw the output on the applet display area. The paint() method executes after the execution of *start()* method and whenever the applet or browser is resized.

The method execution sequence when an applet is executed is:

- init()

- start()
- paint()

The method execution sequence when an applet is closed is:

- stop()
- destroy()
- Example program that demonstrates the life cycle of an applet is as follows:

```
import java.awt.*;
import java.applet.*;
public class MyApplet extends Applet
{
   public void init()
   {
                  System.out.println("Applet initialized");
   }
   public void start()
   {
                  System.out.println("Applet execution started");
   }
   public void stop()
   {
                  System.out.println("Applet execution stopped");
   }
   public void paint(Graphics g)
   {
                  System.out.println("Painting...");
   }
   public void destroy()
   {
                  System.out.println("Applet destroyed");
   }
}
```

- Output of the above applet program when run using appletviewer tool is:

```
Applet initialized
Applet execution started
Painting…
Painting…
Applet execution stopped
Applet destroyed
```

Displaying Image in Applet

Applet is mostly used in games and animation. For this purpose image is required to be displayed. The java.awt.Graphics class provide a method drawImage() to display the image.

Syntax of drawImage() method:

1. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.

How to get the object of Image:

The java.applet.Applet class provides getImage() method that returns the object of Image. Syntax:

```
public Image getImage(URL u, String image){}
```

Other required methods of Applet class to display image:

1. **public URL getDocumentBase():** is used to return the URL of the document in which applet is embedded.
2. **public URL getCodeBase():** is used to return the base URL.

Example of displaying image in applet:

```
import java.awt.*;
import java.applet.*;


public class DisplayImage extends Applet {

  Image picture;

  public void init() {
    picture = getImage(getDocumentBase(),"sonoo.jpg");
  }

  public void paint(Graphics g) {
    g.drawImage(picture, 30,30, this);
  }

  }
```

In the above example, drawImage() method of Graphics class is used to display the image. The 4th argument of drawImage() method of is ImageObserver object. The Component class implements ImageObserver interface. So current class object would also be treated as ImageObserver because Applet class indirectly extends the Component class.

myapplet.html

```
<html>
<body>
<applet code="DisplayImage.class" width="300" height="300">
</applet>
</body>
</html>
```

Adding sound to an applet:

Problem Description

How to play sound using Applet?

Solution

Following example demonstrates how to play a sound using an applet image using getAudioClip(), play() & stop() methods of AudioClip() class.

```java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class PlaySoundApplet extends Applet implements ActionListener {
  Button play,stop;
  AudioClip audioClip;

  public void init() {
    play = new Button(" Play in Loop ");
    add(play);
    play.addActionListener(this);
    stop = new Button(" Stop ");
    add(stop);
    stop.addActionListener(this);
    audioClip = getAudioClip(getCodeBase(), "Sound.wav");
  }
  public void actionPerformed(ActionEvent ae) {
    Button source = (Button)ae.getSource();
    if (source.getLabel() == " Play in Loop ") {
      audioClip.play();
    } else if(source.getLabel() == "  Stop  "){
      audioClip.stop();
    }
  }
}
```

Result

The above code sample will produce the following result in a java enabled web browser.
View in Browser.

Passing parameter to an applet:

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

```java
public String getParameter(String parameterName)
```

Example of using parameter in Applet:

```java
import java.applet.Applet;
import java.awt.Graphics;

public class UseParam extends Applet{

public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}

}
```

myapplet.html

```html
<html>
```

```
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```

Event handling:

EventHandling in Applet

As we perform event handling in AWT or Swing, we can perform it in applet also. Let's see the simple
in applet that prints a message by click on the button.

Example of EventHandling in applet:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class EventApplet extends Applet implements ActionListener{
Button b;
TextField tf;

public void init(){
tf=new TextField();
tf.setBounds(30,40,150,20);

b=new Button("Click");
b.setBounds(80,150,60,50);

add(b);add(tf);
b.addActionListener(this);

setLayout(null);
}

 public void actionPerformed(ActionEvent e){
  tf.setText("Welcome");
 }
}
```

In the above example, we have created all the controls in init() method because
it is invoked only once.

myapplet.html

```
<html>
<body>
<applet code="EventApplet.class" width="300" height="300">
</applet>
</body>
</html>
```

Introducing AWT:

Java AWT Tutorial

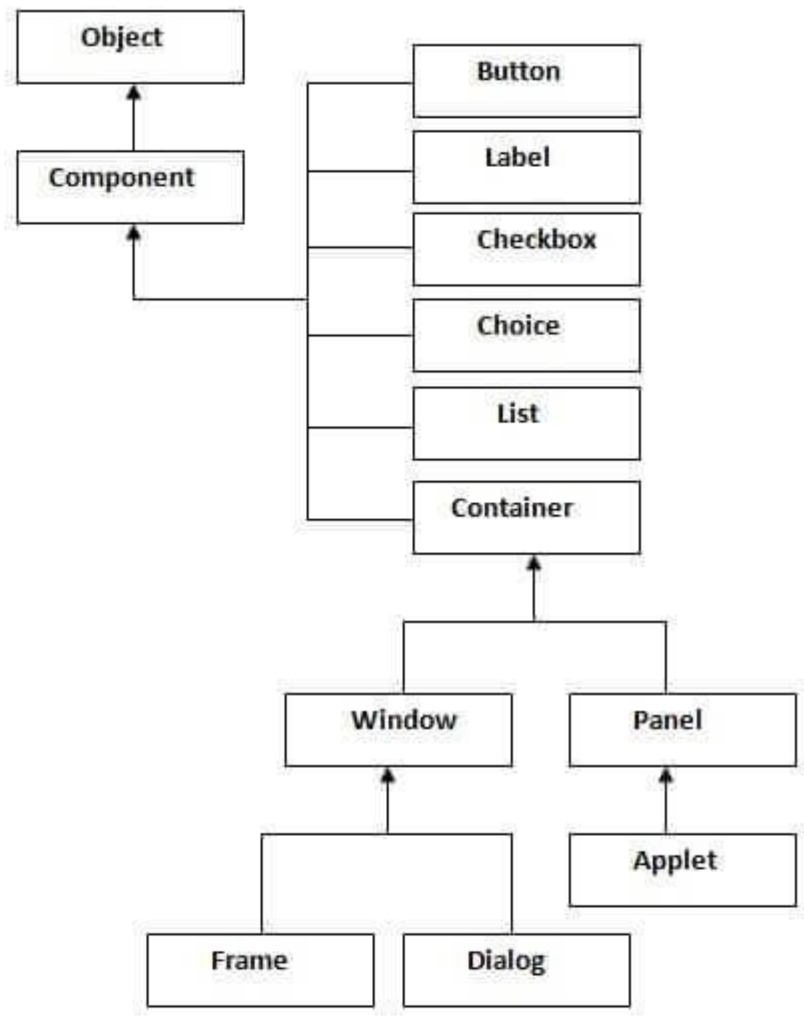
Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

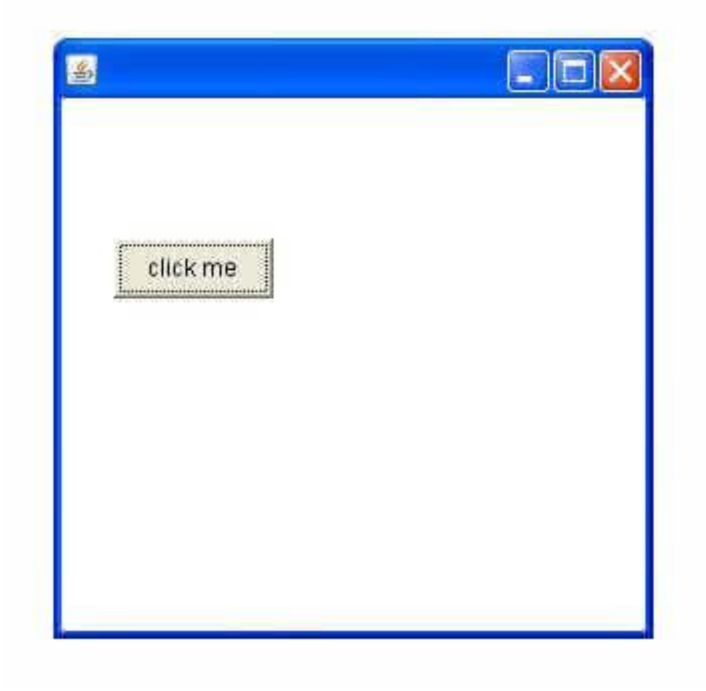
AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```java
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}}
```

download this example

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

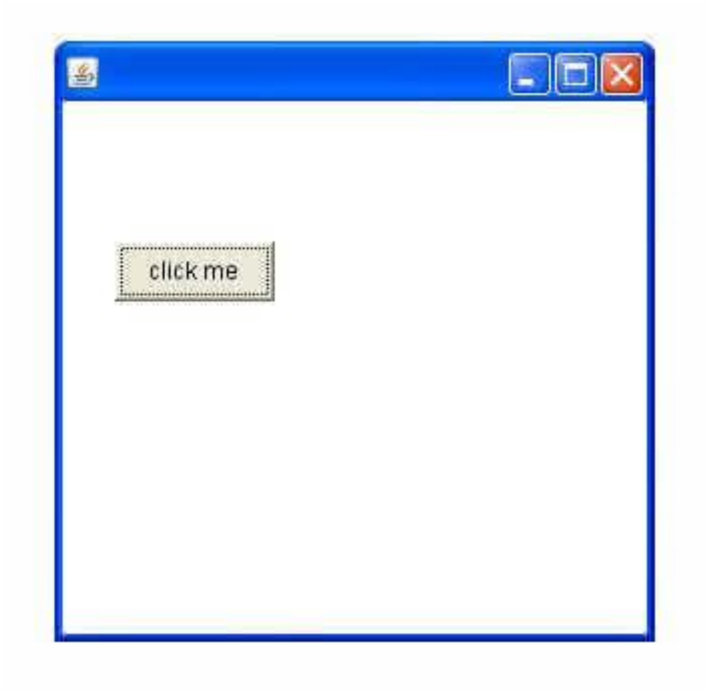


AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First2{
First2(){
Frame f=new Frame();
Button b=new Button("click me");
b.setBounds(30,50,80,30);
f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}}
```

download this example



Working with Windows Graphics and Text.

Introduction

The Graphics class is the abstract super class for all graphics contexts which allow an application to draw onto components that can be realized on various devices, or onto off-screen images as well.

A Graphics object encapsulates all state information required for the basic rendering operations that Java supports. State information includes the following properties.

- The Component object on which to draw.
- A translation origin for rendering and clipping coordinates.
- The current clip.
- The current color.
- The current font.
- The current logical pixel operation function.
- The current XOR alternation color

Class declaration

Following is the declaration for **java.awt.Graphics** class:

```
public abstract class Graphics
  extends Object
```

Class constructors

S.N.	Constructor & Description
1	**Graphics() ()** Constructs a new Graphics object.

Class methods

S.N.	Method & Description
1	**abstract void clearRect(int x, int y, int width, int height)** Clears the specified rectangle by filling it with the background color of the current drawing surface.
2	**abstract void clipRect(int x, int y, int width, int height)** Intersects the current clip with the specified rectangle.
3	**abstract void copyArea(int x, int y, int width, int height, int dx, int dy)** Copies an area of the component by a distance specified by dx and dy.
4	**abstract Graphics create()** Creates a new Graphics object that is a copy of this Graphics object.
5	**Graphics create(int x, int y, int width, int height)** Creates a new Graphics object based on this Graphics object, but with a new translation and clip area.
6	**abstract void dispose()** Disposes of this graphics context and releases any system resources that it is using.
7	**void draw3DRect(int x, int y, int width, int height, boolean raised)** Draws a 3-D highlighted outline of the specified rectangle.
8	**abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)** Draws the outline of a circular or elliptical arc covering the specified rectangle.
9	**void drawBytes(byte[] data, int offset, int length, int x, int y)** Draws the text given by the specified byte array, using this graphics context's current font and color.
10	**void drawChars(char[] data, int offset, int length, int x, int y)** Draws the text given by the specified character array, using this graphics context's current font and color.
11	**abstract boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)** Draws as much of the specified image as is currently available.
12	**abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)** Draws as much of the specified image as is currently available.
13	**abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)** Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.
14	**abstract boolean drawImage(Image img, int x, int y, int width, int height,**

	ImageObserver observer) Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.
15	**abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer)** Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.
16	**abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)** Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.
17	**abstract void drawLine(int x1, int y1, int x2, int y2)** Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system.
18	**abstract void drawOval(int x, int y, int width, int height)** Draws the outline of an oval.
19	**abstract void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)** Draws a closed polygon defined by arrays of x and y coordinates.
20	**void drawPolygon(Polygon p)** Draws the outline of a polygon defined by the specified Polygon object.
21	**abstract void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)** Draws a sequence of connected lines defined by arrays of x and y coordinates.
22	**void drawRect(int x, int y, int width, int height)** Draws the outline of the specified rectangle.
23	**abstract void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)** Draws an outlined round-cornered rectangle using this graphics context's current color.
24	**abstract void drawString(AttributedCharacterIterator iterator, int x, int y)** Renders the text of the specified iterator applying its attributes in accordance with the specification of the TextAttribute class.
25	**abstract void drawString(String str, int x, int y)** Draws the text given by the specified string, using this graphics context's current font and color.
26	**void fill3DRect(int x, int y, int width, int height, boolean raised)** Paints a 3-D highlighted rectangle filled with the current color.
27	**abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)** Fills a circular or elliptical arc covering the specified rectangle.

28	**abstract void fillOval(int x, int y, int width, int height)** Fills an oval bounded by the specified rectangle with the current color.
29	**abstract void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)** Fills a closed polygon defined by arrays of x and y coordinates.
30	**void fillPolygon(Polygon p)** Fills the polygon defined by the specified Polygon object with the graphics context's current color.
31	**abstract void fillRect(int x, int y, int width, int height)** Fills the specified rectangle.
32	**abstract void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)** Fills the specified rounded corner rectangle with the current color.
33	**void finalize()** Disposes of this graphics context once it is no longer referenced.
34	**abstract Shape getClip()** Gets the current clipping area.
35	**abstract Rectangle getClipBounds()** Returns the bounding rectangle of the current clipping area.
36	**Rectangle getClipBounds(Rectangle r)** Returns the bounding rectangle of the current clipping area.
37	**Rectangle getClipRect()** Deprecated. As of JDK version 1.1, replaced by getClipBounds().
38	**abstract Color getColor()** Gets this graphics context's current color.
39	**abstract Font getFont()** Gets the current font.
40	**FontMetrics getFontMetrics()** Gets the font metrics of the current font.
41	**abstract FontMetrics getFontMetrics(Font f)** Gets the font metrics for the specified font.
42	**boolean hitClip(int x, int y, int width, int height)** Returns true if the specified rectangular area might intersect the current clipping area.
43	**abstract void setClip(int x, int y, int width, int height)** Sets the current clip to the rectangle specified by the given coordinates.

44	**abstract void setClip(Shape clip)** Sets the current clipping area to an arbitrary clip shape.
45	**abstract void setColor(Color c)** Sets this graphics context's current color to the specified color.
46	**abstract void setFont(Font font)** Sets this graphics context's font to the specified font.
47	**abstract void setPaintMode()** Sets the paint mode of this graphics context to overwrite the destination with this graphics context's current color.
48	**abstract void setXORMode(Color c1)** Sets the paint mode of this graphics context to alternate between this graphics context's current color and the new specified color.
49	**String toString()** Returns a String object representing this Graphics object's value.
50	**abstract void translate(int x, int y)** Translates the origin of the graphics context to the point (x, y) in the current coordinate system.

Methods inherited

This class inherits methods from the following classes:

- java.lang.Object

Graphics Example

Create the following java program using any editor of your choice in say **D:/ > AWT > com > tutorialspoint > gui >**

AWTGraphicsDemo.java

```
package com.tutorialspoint.gui;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

public class AWTGraphicsDemo extends Frame {

   public AWTGraphicsDemo(){
      super("Java AWT Examples");
      prepareGUI();
   }

   public static void main(String[] args){
      AWTGraphicsDemo  awtGraphicsDemo = new AWTGraphicsDemo();
      awtGraphicsDemo.setVisible(true);
   }

   private void prepareGUI(){
      setSize(400,400);
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent windowEvent){
            System.exit(0);
         }
      });
   }

   @Override
   public void paint(Graphics g) {
      g.setColor(Color.GRAY);
      Font font = new Font("Serif", Font.PLAIN, 24);
      g.setFont(font);
      g.drawString("Welcome to TutorialsPoint", 50, 150);
   }
}
```

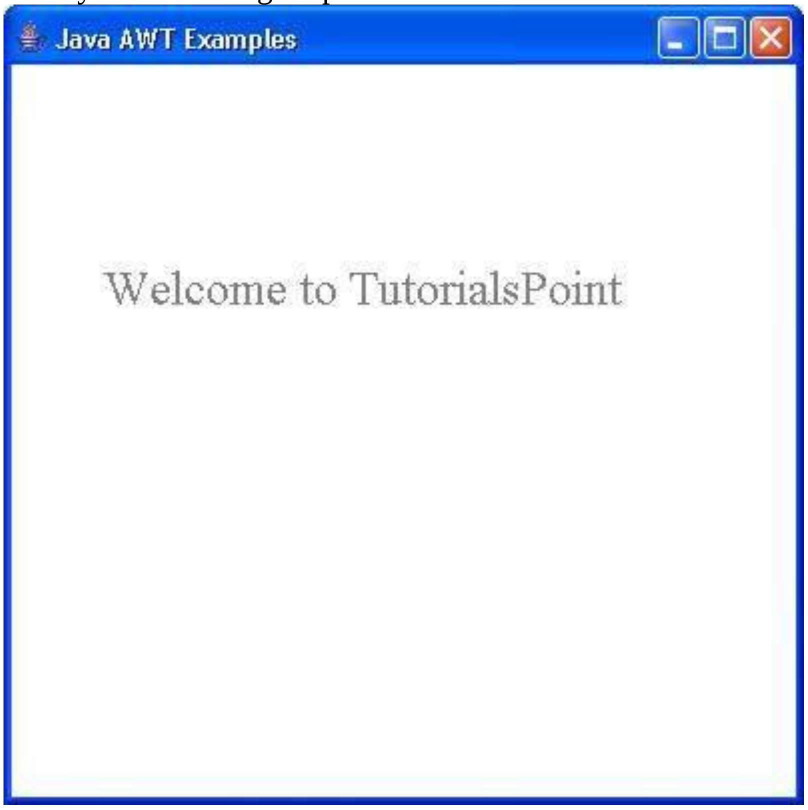
Compile the program using command prompt. Go to **D:/ > AWT** and type the following command.

```
D:\AWT>javac com\tutorialspoint\gui\AWTGraphicsDemo.java
```

If no error comes that means compilation is successful. Run the program using following command.

```
D:\AWT>java com.tutorialspoint.gui.AWTGraphicsDemo
```

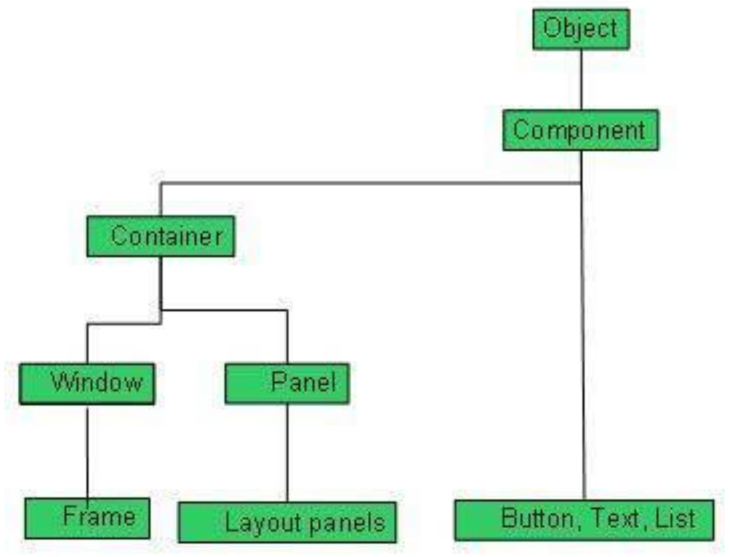
Verify the following output



Welcome to TutorialsPoint

Every user interface considers the following three main aspects:

- **UI elements** : Thes are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex which we will cover in this tutorial.
- **Layouts:** They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in Layout chapter.
- **Behavior:** These are events which occur when the user interacts with UI elements. This part will be covered in Event Handling chapter.



Every AWT controls inherits properties from Component class.

Sr. No.	Control & Description
1	Component A Component is an abstract super class for GUI controls and it represents an object with graphical representation.

AWT UI Elements:

Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	Control & Description
1	Label A Label object is a component for placing text in a container.
2	Button This class creates a labeled button.
3	Check Box A check box is a graphical component that can be in either an **on** (true) or **off** (false) state.
4	Check Box Group The CheckboxGroup class is used to group the set of checkbox.
5	List

	The List component presents the user with a scrolling list of text items.
6	Text Field A TextField object is a text component that allows for the editing of a single line of text.
7	Text Area A TextArea object is a text component that allows for the editing of a multiple lines of text.
8	Choice A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
9	Canvas A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.
10	Image An Image control is superclass for all image classes representing graphical images.
11	Scroll Bar A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
12	Dialog A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.
13	File Dialog A FileDialog control represents a dialog window from which the user can select a file.

Layout Managers and Menus:

Introduction

Layout means the arrangement of components within the container. In other way we can say that placing the components at a particular position within the container. The task of layouting the controls is done automatically by the Layout Manager.

Layout Manager

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

- It is very tedious to handle a large number of controls within the container.
- Oftenly the width and height information of a component is not given when we need to arrange them.

Java provide us with various layout manager to position the controls. The properties like size,shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the LayoutManager interface.
Following are the interfaces defining functionalities of Layout Managers.

Sr. No.	Interface & Description
1	LayoutManager The LayoutManager interface declares those methods which need to be implemented by the class whose object will act as a layout manager.
2	LayoutManager2 The LayoutManager2 is the sub-interface of the LayoutManager.This interface is for those classes that know how to layout containers based on layout constraint object.

AWT Layout Manager Classes:
Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	LayoutManager & Description
1	BorderLayout The borderlayout arranges the components to fit in the five regions: east, west, north, south and center.
2	CardLayout The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
3	FlowLayout The FlowLayout is the default layout.It layouts the components in a directional flow.
4	GridLayout The GridLayout manages the components in form of a rectangular grid.
5	GridBagLayout This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.

Servlet:

Servlets | Servlet Tutorial

Servlet technology is used to create a web application (resides at server side and generates a dynamic web page).
Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.
There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

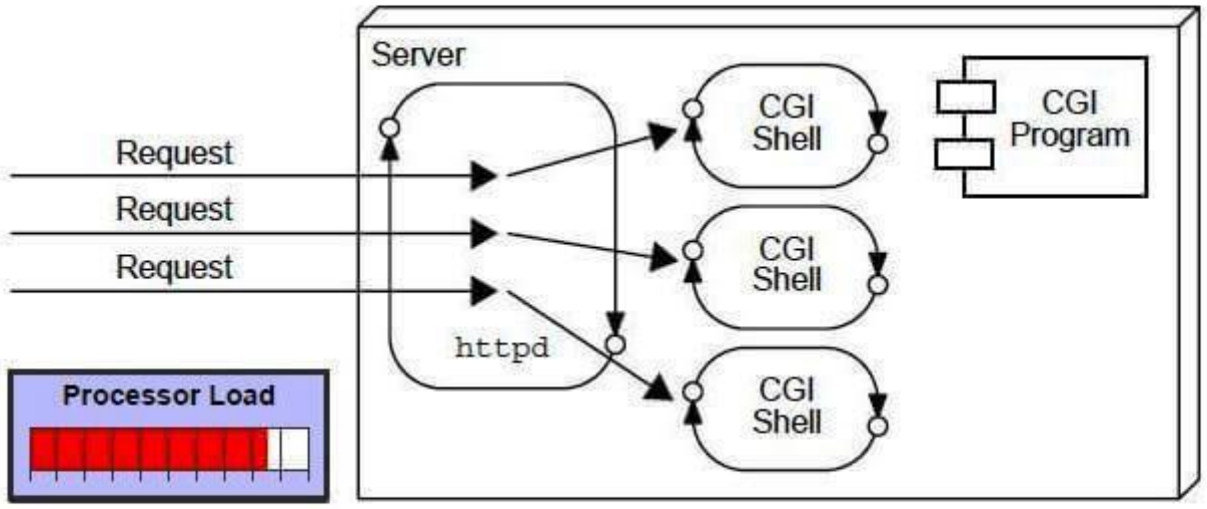
What is a Servlet?

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.

CGI (Common Gateway Interface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

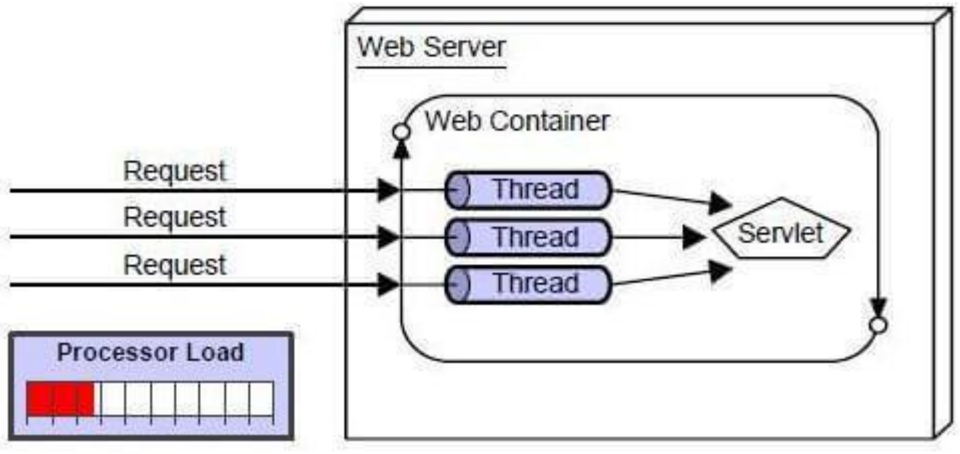


Disadvantages of CGI

There are many problems in CGI technology:

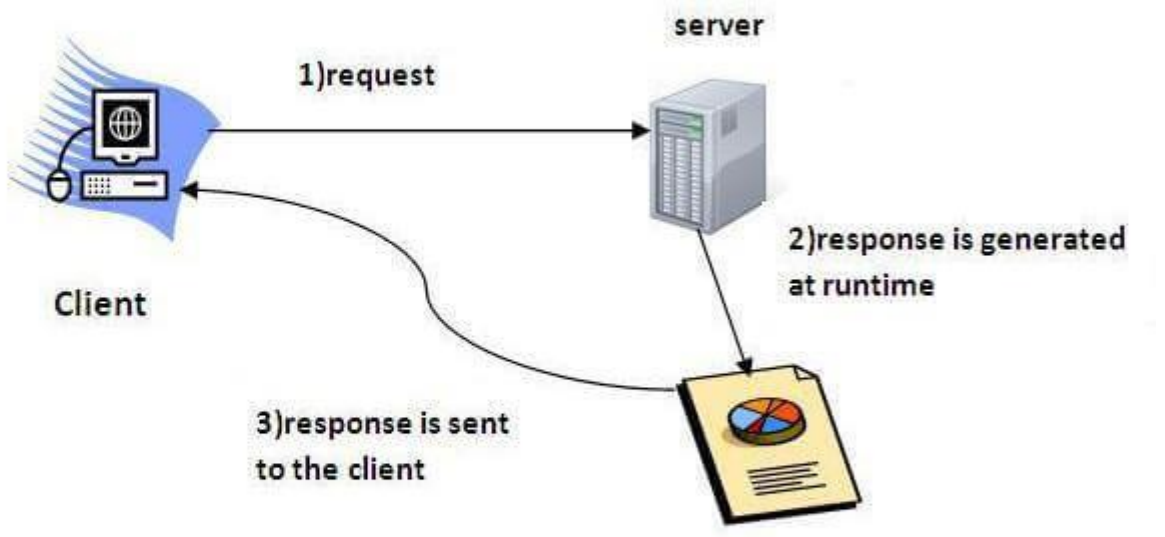
1. If the number of clients increases, it takes more time for sending the response.
2. For each request, it starts a process, and the web server is limited to start processes.
3. It uses platform dependent language e.g. C, C++, perl.

Advantages of Servlet



There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

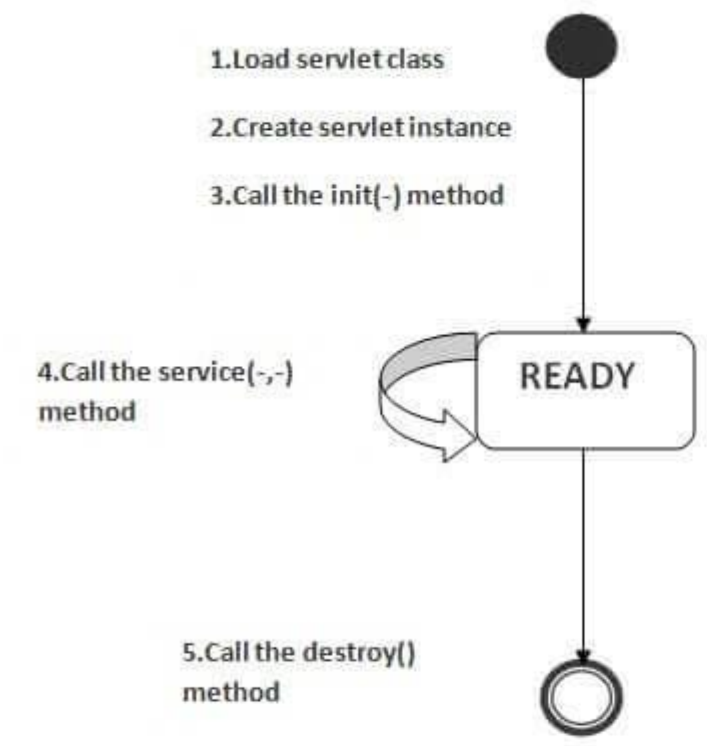
1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.


server
1)request
2)response is generated at runtime
Client
3)response is sent to the client

Life cycle of an Servlet:

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.


1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

---

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the Javax.servlet.Servlet interface. Syntax of the init method is given below:

```
1. public void init(ServletConfig config) throws ServletException
```

4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

```
1. public void service(ServletRequest request, ServletResponse response)
2.   throws ServletException, IOException
```

---

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

```
1. public void destroy()
```

The Servlet API:

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api. The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel
8. Filter
9. FilterConfig
10. FilterChain
11. ServletRequestListener
12. ServletRequestAttributeListener
13. ServletContextListener
14. ServletContextAttributeListener

Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession
4. HttpSessionListener
5. HttpSessionAttributeListener
6. HttpSessionBindingListener
7. HttpSessionActivationListener
8. HttpSessionContext (deprecated now)

Classes in javax.servlet.http package

There are many classes in javax.servlet.http package. They are as follows:

1. HttpServlet
2. Cookie
3. HttpServletRequestWrapper
4. HttpServletResponseWrapper
5. HttpSessionEvent
6. HttpSessionBindingEvent
7. HttpUtils (deprecated now)

Handling HTTP Request and Response:

An HTTP client sends an HTTP request to a server in the form of a request message which includes following format:

- A Request-line
- Zero or more header (General|Request|Entity) fields followed by CRLF
- An empty line (i.e., a line with nothing preceding the CRLF)
- indicating the end of the header fields
- Optionally a message-body

The following sections explain each of the entities used in an HTTP request message.

Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by space SP characters.

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Let's discuss each of the parts mentioned in the Request-Line.

Request Method

The request **method** indicates the method to be performed on the resource identified by the given **Request-URI**. The method is case-sensitive and should always be mentioned in uppercase. The following table lists all the supported methods in HTTP/1.1.

S.N.	Method and Description
1	**GET** The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
2	**HEAD** Same as GET, but it transfers the status line and the header section only.
3	**POST** A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.
4	**PUT** Replaces all the current representations of the target resource with the uploaded content.
5	**DELETE** Removes all the current representations of the target resource given by URI.
6	**CONNECT** Establishes a tunnel to the server identified by a given URI.
7	**OPTIONS** Describe the communication options for the target resource.
8	**TRACE** Performs a message loop back test along with the path to the target resource.

Request-URI

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request. Following are the most commonly used forms to specify an URI:

Request-URI = "*" | absoluteURI | abs_path | authority

S.N.	Method and Description
1	The asterisk ***** is used when an HTTP request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a resource. For example: **OPTIONS * HTTP/1.1**
2	The **absoluteURI** is used when an HTTP request is being made to a proxy. The proxy is requested to forward the request or service from a valid cache, and return the response. For example: **GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1**
3	The most common form of Request-URI is that used to identify a resource on an origin server or gateway. For example, a client wishing to retrieve a resource directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the following lines: **GET /pub/WWW/TheProject.html HTTP/1.1** **Host: www.w3.org** Note that the absolute path cannot be empty; if none is present in the original URI, it MUST be given as "/" (the server root).

Request Header Fields

We will study General-header and Entity-header in a separate chapter when we will learn HTTP header fields. For now, let's check what Request header fields are.

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers.Here is a list of some important Request-header fields that can be used based on the requirement:

- Accept-Charset
- Accept-Encoding
- Accept-Language
- Authorization
- Expect
- From
- Host
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Proxy-Authorization
- Range
- Referer
- TE
- User-Agent

You can introduce your custom fields in case you are going to write your own custom Client and Web Server.

Examples of Request Message

Now let's put it all together to form an HTTP request to fetch **hello.htm** page from the web server running on tutorialspoint.com

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Here we are not sending any request data to the server because we are fetching a plain HTML page from the server. Connection is a general-header, and the rest of the headers are request headers. The following example shows how to send form data to the server using request message body:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string
```

Here the given URL */cgi-bin/process.cgi* will be used to process the passed data and accordingly, a response will be returned. Here **content-type** tells the server that the passed data is a simple web form data and **length** will be the actual length of the data put in the message body. The following example shows how you can pass plain XML to your web server:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: text/xml; charset=utf-8
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://clearforest.com/">string</string>
```

Using Cookies:

In the last guide, I have covered Sessions in Servlet. Here we will discuss Cookies which is also used for session management. Let's recall few things here from last tutorial so that we can relate sessions and cookies. When a user visits web application first time, the servlet container crates new HttpSession object by calling request.getSession(). A unique Id is assigned to the session. The **Servlet container also sets a Cookie in the header of the HTTP response with cookie name and the unique session ID as its value.**

The cookie is stored in the user browser, the client (user's browser) sends this cookie back to the server for all the subsequent requests until the cookie is valid. **The Servlet container checks the request header for cookies and get the session information from the cookie and use the associated session from the server memory.**
The session remains active for the time specified in tag in web.xml. If tag in not set in web.xml then the session remains active for 30 minutes. **Cookie remains active as long as the user's browser is running**, as soon as the browser is closed, the cookie and associated session info is destroyed. So when the user opens the browser again and sends request to web server, the new session is being created.

Types of Cookies

We can classify the cookie based on their expiry time:

1. Session
2. Persistent

1) SessionCookies:

Session cookies do not have expiration time. It lives in the browser memory. As soon as the web browser is closed this cookie gets destroyed.

2) Persistent Cookies:

Unlike Session cookies they have expiration time, they are stored in the user hard drive and gets destroyed based on the expiry time.

How to send Cookies to the Client

Here are steps for sending cookie to the client:

1. Create a Cookie object.
2. Set the maximum Age.
3. Place the Cookie in HTTP response header.

1) Create a Cookie object:

```
Cookie c = new Cookie("userName","Chaitanya");
```

2) Set the maximum Age:

By using **setMaxAge ()** method we can set the maximum age for the particular cookie in seconds.

```
c.setMaxAge(1800);
```

3) Place the Cookie in HTTP response header:

We can send the cookie to the client browser through response.addCookie() method.

```
response.addCookie(c);
```

How to read cookies

```
Cookie c[]=request.getCookies();
//c.length gives the cookie count
for(int i=0;i<c.length;i++){
 out.print("Name: "+c[i].getName()+" & Value: "+c[i].getValue());
}
```

Example of Cookies in java servlet

index.html

```
<form action="login">
 User Name:<input type="text" name="userName"/><br/>
 Password:<input type="password" name="userPassword"/><br/>
 <input type="submit" value="submit"/>
</form>
```

MyServlet1.java

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

```
public class MySerlet1 extends HttpServlet
{
  public void doGet(HttpServletRequest request,
    HttpServletResponse response) {
    try{
        response.setContentType("text/html");
        PrintWriter pwriter = response.getWriter();

        String name = request.getParameter("userName");
        String password = request.getParameter("userPassword");
        pwriter.print("Hello "+name);
        pwriter.print("Your Password is: "+password);

        //Creating two cookies
        Cookie c1=new Cookie("userName",name);
        Cookie c2=new Cookie("userPassword",password);

        //Adding the cookies to response header
        response.addCookie(c1);
        response.addCookie(c2);
        pwriter.print("<br><a href='welcome'>View Details</a>");
        pwriter.close();
  }catch(Exception exp){
      System.out.println(exp);
   }
  }
}
```

MyServlet2.java

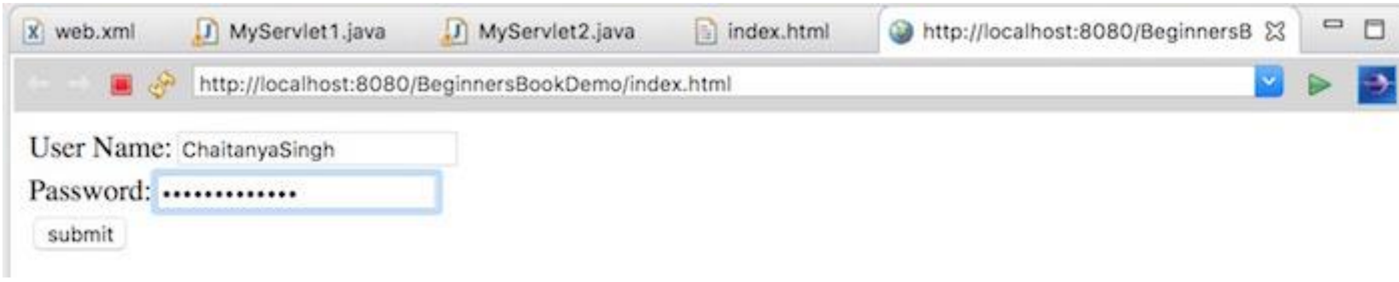
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet2 extends HttpServlet {
 public void doGet(HttpServletRequest request,
   HttpServletResponse response){
   try{
      response.setContentType("text/html");
      PrintWriter pwriter = response.getWriter();

      //Reading cookies
      Cookie c[]=request.getCookies();
      //Displaying User name value from cookie
      pwriter.print("Name: "+c[1].getValue());
      //Displaying user password value from cookie
      pwriter.print("Password: "+c[2].getValue());

      pwriter.close();
   }catch(Exception exp){
      System.out.println(exp);
    }
  }
}
```

web.xml

```
<web-app>
<display-name>BeginnersBookDemo</display-name>
 <welcome-file-list>
 <welcome-file>index.html</welcome-file>
 </welcome-file-list>
<servlet>
 <servlet-name>Servlet1</servlet-name>
 <servlet-class>MyServlet1</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Servlet1</servlet-name>
 <url-pattern>/login</url-pattern>
</servlet-mapping>
<servlet>
 <servlet-name>Servlet2</servlet-name>
 <servlet-class>MyServlet2</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Servlet2</servlet-name>
 <url-pattern>/welcome</url-pattern>
</servlet-mapping>
</web-app>
```

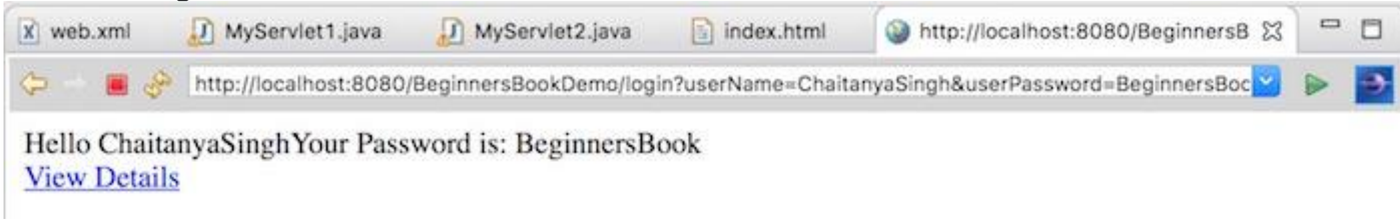
Output:

Welcome Screen:


web.xml
MyServlet1.java
MyServlet2.java
index.html
http://localhost:8080/BeginnersB
http://localhost:8080/BeginnersBookDemo/index.html
submit

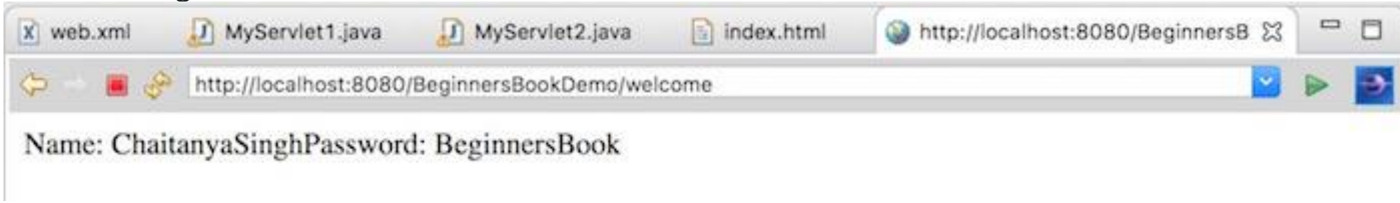
User Name: ChaitanyaSingh
Password: ••••••••••••••

After clicking Submit:


web.xml
MyServlet1.java
MyServlet2.java
index.html
http://localhost:8080/BeginnersB
http://localhost:8080/BeginnersBookDemo/login?userName=ChaitanyaSingh&userPassword=BeginnersBoc

Hello ChaitanyaSinghYour Password is: BeginnersBook
View Details

After clicking View Details:


web.xml
MyServlet1.java
MyServlet2.java
index.html
http://localhost:8080/BeginnersB
http://localhost:8080/BeginnersBookDemo/welcome

Name: ChaitanyaSinghPassword: BeginnersBook

Methods of Cookie class

public void setComment(String purpose): This method is used for setting up comments in the cookie. This is basically used for describing the purpose of the cookie.

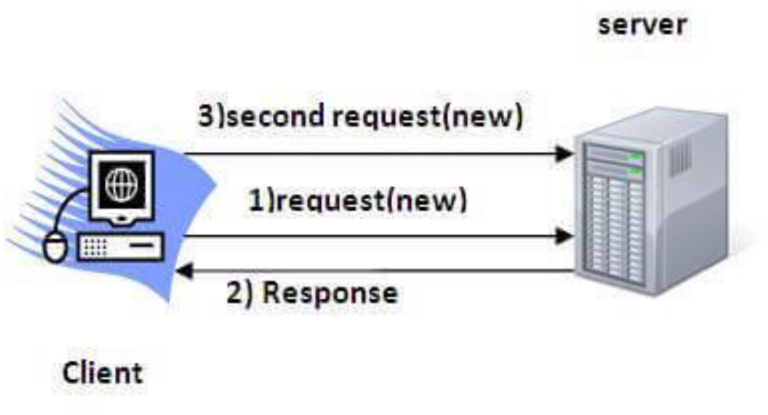
public String getComment(): Returns the comment describing the purpose of this cookie, or null if the cookie has no comment.
public void setMaxAge(int expiry): Sets the maximum age of the cookie in seconds.
public int getMaxAge(): Gets the maximum age in seconds of this Cookie.
By default, -1 is returned, which indicates that the cookie will persist until browser shutdown.
public String getName(): Returns the name of the cookie. The name cannot be changed after creation.
public void setValue(String newValue): Assigns a new value to this Cookie.
public String getValue(): Gets the current value of this Cookie.
The list above has only commonly used methods. To get the complete list of methods of Cookie class refer official documentation.

Session Tracking:

Session Tracking in Servlets

1. Session Tracking
2. Session Tracking Techniques

Session simply means a particular interval of time.
Session Tracking is a way to maintain state (data) of an user. It is also known as **session management** in servlet.
Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.
HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:



Why use Session Tracking?

To recognize the user It is used to recognize the particular user.

Session Tracking Techniques

There are four techniques used in Session tracking:

1. **Cookies**
2. **Hidden Form Field**
3. **URL Rewriting**
4. **HttpSession**

Introduction to JSP:

What is JavaServer Pages?

JavaServer Pages (JSP) is a technology for developing Webpages that supports dynamic content. This helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.

A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.
Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically.
JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages, and sharing information between requests, pages etc.

Why Use JSP?

JavaServer Pages often serve the same purpose as programs implemented using the **Common Gateway Interface (CGI)**. But JSP offers several advantages in comparison with the CGI.

- Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having separate CGI files.
- JSP are always compiled before they are processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.
- JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including **JDBC, JNDI, EJB, JAXP,** etc.
- JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

Advantages of JSP

Following table lists out the other advantages of using JSP over other technologies −

vs. Active Server Pages (ASP)

The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.

vs. Pure Servlets

It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.

vs. Server-Side Includes (SSI)

SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

vs. JavaScript

JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.

vs. Static HTML

Regular HTML, of course, cannot contain dynamic information.

Unit-V

XML AND WEB SERVICES

Xml – Introduction-Form Navigation-XML Documents- XSL – XSLT- Web services-UDDI-WSDL-Java web services – Web resources.

XML:

XML stands for **E**xtensible **M**arkup **L**anguage. It is a text-based markup language derived from Standard Generalized Markup Language (SGML).

XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.

There are three important characteristics of XML that make it useful in a variety of systems and solutions −

- **XML is extensible** − XML allows you to create your own self-descriptive tags, or language, that suits your application.
- **XML carries the data, does not present it** − XML allows you to store the data irrespective of how it will be presented.
- **XML is a public standard** − XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

XML Usage

A short list of XML usage says it all −

- XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- XML can be used to exchange the information between organizations and systems.
- XML can be used for offloading and reloading of databases.
- XML can be used to store and arrange the data, which can customize your data handling needs.
- XML can easily be merged with style sheets to create almost any desired output.
- Virtually, any type of data can be expressed as an XML document.

What is Markup?

XML is a markup language that defines set of rules for encoding documents in a format that is both human-readable and machine-readable. So *what exactly is a markup language?* Markup is information added to a document that enhances its meaning in certain ways, in that it identifies the parts and how they relate to each other. More specifically, a markup language is a set of symbols that can be placed in the text of a document to demarcate and label the parts of that document.

Following example shows how XML markup looks, when embedded in a piece of text −

```
<message>
  <text>Hello, world!</text>
</message>
```

This snippet includes the markup symbols, or the tags such as <message>...</message> and <text>... </text>. The tags <message> and </message> mark the start and the end of the XML code fragment. The tags <text> and </text> surround the text Hello, world!.

Is XML a Programming Language?

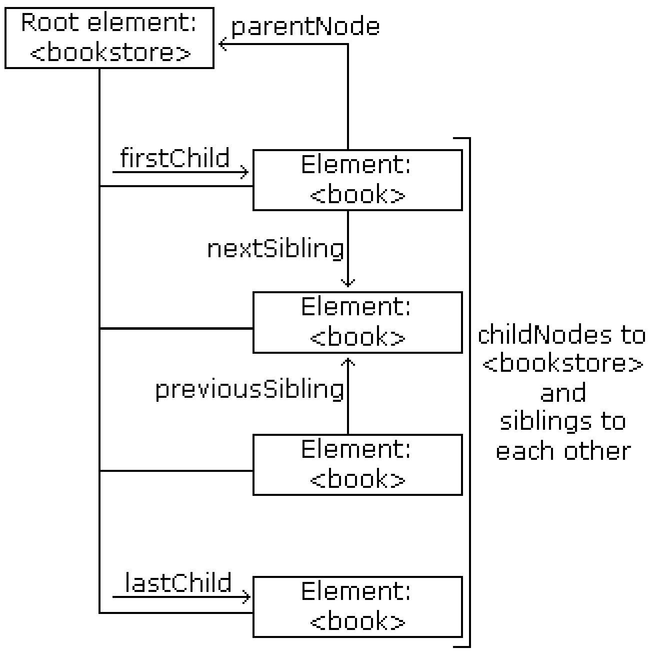
A programming language consists of grammar rules and its own vocabulary which is used to create computer programs. These programs instruct the computer to perform specific tasks. XML does not qualify to be a programming language as it does not perform any computation or algorithms. It is usually stored in a simple text file and is processed by special software that is capable of interpreting XML.

Form Navigation:

Navigating DOM Nodes
Accessing nodes in the node tree via the relationship between nodes, is often called "navigating nodes".
In the XML DOM, node relationships are defined as properties to the nodes:

- parentNode
- childNodes
- firstChild
- lastChild
- nextSibling
- previousSibling

The following image illustrates a part of the node tree and the relationship between nodes in books.xml:



DOM - Parent Node
All nodes have exactly one parent node. The following code navigates to the parent node of <book>:

Example
```
function myFunction(xml) {
var xmlDoc = xml.responseXML;
   var x = xmlDoc.getElementsByTagName("book")[0];
   document.getElementById("demo").innerHTML = x.parentNode.nodeName;
}
```
Avoid Empty Text Nodes
Firefox, and some other browsers, will treat empty white-spaces or new lines as text nodes, Internet Explorer will not.
This causes a problem when using the properties: firstChild, lastChild, nextSibling, previousSibling.

To avoid navigating to empty text nodes (spaces and new-line characters between element nodes), we use a function that checks the node type:

```
function get_nextSibling(n) {
    var y = n.nextSibling;
    while (y.nodeType! = 1) {
        y = y.nextSibling;
    }
    return y;
}
```

The function above allows you to use get_nextSibling(*node*) instead of the property *node*.nextSibling.

Code explained:

Element nodes are type 1. If the sibling node is not an element node, it moves to the next nodes until an element node is found. This way, the result will be the same in both Internet Explorer and Firefox.

Get the First Child Element

The following code displays the first element node of the first <book>:

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myFunction(this);
    }
};
xhttp.open("GET", "books.xml", true);
xhttp.send();

function myFunction(xml) {
    var xmlDoc = xml.responseXML;
    var x = get_firstChild(xmlDoc.getElementsByTagName("book")[0]);
    document.getElementById("demo").innerHTML = x.nodeName;
}

//check if the first node is an element node
function get_firstChild(n) {
    var y = n.firstChild;
    while (y.nodeType != 1) {
        y = y.nextSibling;
    }
    return y;
}
</script>
```

```
</body>
</html>
```

Output:

title

XML Documents:

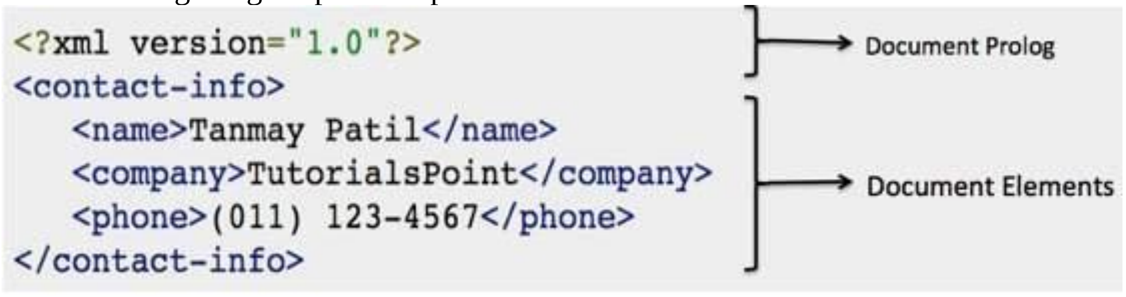
An XML *document* is a basic unit of XML information composed of elements and other markup in an orderly package. An XML *document* can contains wide variety of data. For example, database of numbers, numbers representing molecular structure or a mathematical equation.

XML Document Example

A simple document is shown in the following example −

```
<?xml version = "1.0"?>
<contact-info>
  <name>Tanmay Patil</name>
  <company>TutorialsPoint</company>
  <phone>(011) 123-4567</phone>
</contact-info>
```

The following image depicts the parts of XML document.



```
<?xml version="1.0"?>
<contact-info>
   <name>Tanmay Patil</name>
   <company>TutorialsPoint</company>
   <phone>(011) 123-4567</phone>
</contact-info>
```

Document Prolog Section

Document Prolog comes at the top of the document, before the root element. This section contains −

- XML declaration
- Document type declaration

You can learn more about XML declaration in this chapter − XML Declaration

Document Elements Section

Document Elements are the building blocks of XML. These divide the document into a hierarchy of sections, each serving a specific purpose. You can separate a document into multiple sections so that they can be rendered differently, or used by a search engine. The elements can be containers, with a combination of text and other elements.

XSLT:

XSL (eXtensible Stylesheet Language) is a styling language for XML.
XSLT stands for XSL Transformations.
This tutorial will teach you how to use XSLT to transform XML documents into other formats (like transforming XML into HTML).

Online XSLT Editor

With our online editor, you can edit XML and XSLT code, and click on a button to view the result.

XSLT Example
```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <xsl:for-each select="catalog/cd">
        <tr>
          <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="artist"/></td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```
What You Should Already Know
Before you continue you should have a basic understanding of the following:

- HTML
- XML

If you want to study these subjects first, find the tutorials on our Home page.
XSLT References
XSLT Elements
Description of all the XSLT elements from the W3C Recommendation, and information about browser support.
XSLT, XPath, and XQuery Functions
XSLT 2.0, XPath 2.0, and XQuery 1.0, share the same functions library. There are over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, and more.

Web Services:
Web services are web application components.
Web services can be published, found, and used on the Web.
This tutorial introduces WSDL, SOAP, RDF, and RSS.

WSDL

- WSDL stands for Web Services Description Language
- WSDL is an XML-based language for describing Web services.
- WSDL is a W3C recommendation

SOAP

- SOAP stands for Simple Object Access Protocol
- SOAP is an XML based protocol for accessing Web Services.
- SOAP is based on XML
- SOAP is a W3C recommendation

RDF

- RDF stands for Resource Description Framework
- RDF is a framework for describing resources on the web
- RDF is written in XML
- RDF is a W3C Recommendation

RSS

- RSS stands for Really Simple Syndication
- RSS allows you to syndicate your site content
- RSS defines an easy way to share and view headlines and content
- RSS files can be automatically updated
- RSS allows personalized views for different sites
- RSS is written in XML

What You Should Already Know

Before you study web services you should have a basic understanding of XML and XML Namespaces.

If you want to study these subjects first, please read our XML Tutorial.

Web Services

- Web services are application components
- Web services communicate using open protocols
- Web services are self-contained and self-describing
- Web services can be discovered using UDDI
- Web services can be used by other applications
- HTTP and XML is the basis for Web services

Interoperability has Highest Priority

When all major platforms could access the Web using Web browsers, different platforms couldn't interact. For these platforms to work together, Web-applications were developed.

Web-applications are simply applications that run on the web. These are built around the Web browser standards and can be used by any browser on any platform.

Web Services take Web-applications to the Next Level

By using Web services, your application can publish its function or message to the rest of the world.

Web services use XML to code and to decode data, and SOAP to transport it (using open protocols).

With Web services, your accounting department's Win 2k server's billing system can connect with your IT supplier's UNIX server.

Web Services have Two Types of Uses

Reusable application-components.

There are things applications need very often. So why make these over and over again?

Web services can offer application-components like: currency conversion, weather reports, or even language translation as services.

Connect existing software.
Web services can help to solve the interoperability problem by giving different applications a way to link their data.
With Web services you can exchange data between different applications and different platforms.
Any application can have a Web Service component.
Web Services can be created regardless of programming language.

A Web Service Example
In the following example we will use ASP.NET to create a simple Web Service that converts the temperature from Fahrenheit to Celsius, and vice versa:

```
<%@ WebService Language="VBScript" Class="TempConvert" %>

Imports System
Imports System.Web.Services

Public Class TempConvert :Inherits WebService

<WebMethod()> Public Function FahrenheitToCelsius(ByVal Fahrenheit As String) As
String
  dim fahr
  fahr=trim(replace(Fahrenheit,",","."))
  if fahr="" or IsNumeric(fahr)=false then return "Error"
  return ((((fahr) - 32) / 9) * 5)
end function

<WebMethod()> Public Function CelsiusToFahrenheit(ByVal Celsius As String) As String
  dim cel
  cel=trim(replace(Celsius,",","."))
  if cel="" or IsNumeric(cel)=false then return "Error"
  return ((((cel) * 9) / 5) + 32)
end function

end class
```

This document is saved as an .asmx file. This is the ASP.NET file extension for XML Web Services.

Example Explained
Note: To run this example, you will need a .NET server.
The first line in the example states that this is a Web Service, written in VBScript, and has the class name "TempConvert":

```
<%@ WebService Language="VBScript" Class="TempConvert" %>
```

The next lines import the namespace "System.Web.Services" from the .NET framework:

```
Imports System
Imports System.Web.Services
```

The next line defines that the "TempConvert" class is a WebService class type:

```
Public Class TempConvert :Inherits WebService
```

The next steps are basic VB programming. This application has two functions. One to convert from Fahrenheit to Celsius, and one to convert from Celsius to Fahrenheit.

The only difference from a normal application is that this function is defined as a "WebMethod()".
Use "WebMethod()" to convert the functions in your application into web services:

```
<WebMethod()> Public Function FahrenheitToCelsius(ByVal Fahrenheit As String) As
String
  dim fahr
  fahr=trim(replace(Fahrenheit,",","."))
  if fahr="" or IsNumeric(fahr)=false then return "Error"
  return ((((fahr) - 32) / 9) * 5)
end function

<WebMethod()> Public Function CelsiusToFahrenheit(ByVal Celsius As String) As String
  dim cel
  cel=trim(replace(Celsius,",","."))
  if cel="" or IsNumeric(cel)=false then return "Error"
  return ((((cel) * 9) / 5) + 32)
end function
```

Then, end the class:

```
end class
```

Publish the .asmx file on a server with .NET support, and you will have your first working Web Service.

Put the Web Service on Your Web Site

Using a form and the HTTP POST method, you can put the web service on your site, like this:

Fahrenheit to Celsius: Submit

Celsius to Fahrenheit: Submit

How To Do It

Here is the code to add the Web Service to a web page:

```
<form action='tempconvert.asmx/FahrenheitToCelsius'
method="post" target="_blank">
<table>
  <tr>
    <td>Fahrenheit to Celsius:</td>
    <td>
    <input class="frmInput" type="text" size="30" name="Fahrenheit">
    </td>
  </tr>
  <tr>
    <td></td>
    <td align="right">
     <input type="submit" value="Submit" class="button">
     </td>
  </tr>
</table>
```

```
</form>

<form action='tempconvert.asmx/CelsiusToFahrenheit'
method="post" target="_blank">
<table>
  <tr>
    <td>Celsius to Fahrenheit:</td>
    <td>
    <input class="frmInput" type="text" size="30" name="Celsius">
    </td>
  </tr>
  <tr>
    <td></td>
    <td align="right">
    <input type="submit" value="Submit" class="button">
    </td>
  </tr>
</table>
</form>
```

UDDI:

UDDI is an XML-based standard for describing, publishing, and finding web services.

- UDDI stands for **Universal Description, Discovery, and Integration.**
- UDDI is a specification for a distributed registry of web services.
- UDDI is a platform-independent, open framework.
- UDDI can communicate via SOAP, CORBA, Java RMI Protocol.
- UDDI uses Web Service Definition Language(WSDL) to describe interfaces to web services.
- UDDI is seen with SOAP and WSDL as one of the three foundation standards of web services.
- UDDI is an open industry initiative, enabling businesses to discover each other and define how they interact over the Internet.

UDDI has two sections −

- A registry of all web service's metadata, including a pointer to the WSDL description of a service.
- A set of WSDL port type definitions for manipulating and searching that registry.

History of UDDI

- UDDI 1.0 was originally announced by Microsoft, IBM, and Ariba in September 2000.
- Since the initial announcement, the UDDI initiative has grown to include more than 300 companies including Dell, Fujitsu, HP, Hitachi, IBM, Intel, Microsoft, Oracle, SAP, and Sun.
- In May 2001, Microsoft and IBM launched the first UDDI operator sites and turned the UDDI registry live.
- In June 2001, UDDI announced Version 2.0.
- As the time of writing this tutorial, Microsoft and IBM sites had implemented the 1.0 specification and were planning 2.0 support in the near future.
- Currently UDDI is sponsored by OASIS.

Partner Interface Processes

Partner Interface Processes (PIPs) are XML based interfaces that enable two trading partners to exchange data. Dozens of PIPs already exist. Some of them are listed here −

- **PIP2A2** − Enables a partner to query another for product information.
- **PIP3A2** − Enables a partner to query the price and availability of specific products.
- **PIP3A4** − Enables a partner to submit an electronic purchase order and receive acknowledgment of the order.
- **PIP3A3** − Enables a partner to transfer the contents of an electronic shopping cart.
- **PIP3B4** − Enables a partner to query the status of a specific shipment.

Private UDDI Registries

As an alternative to using the public federated network of UDDI registries available on the Internet, companies or industry groups may choose to implement their own private UDDI registries.

These exclusive services are designed for the sole purpose of allowing members of the company or of the industry group to share and advertise services amongst themselves.

Regardless of whether the UDDI registry is a part of the global federated network or a privately owned and operated registry, the one thing that ties them all together is a common web services API for publishing and locating businesses and services advertised within the UDDI registry.

WSDL:

WSDL stands for Web Services Description Language. It is the standard format for describing a web service. WSDL was developed jointly by Microsoft and IBM.

Features of WSDL

- WSDL is an XML-based protocol for information exchange in decentralized and distributed environments.
- WSDL definitions describe how to access a web service and what operations it will perform.
- WSDL is a language for describing how to interface with XML-based services.
- WSDL is an integral part of Universal Description, Discovery, and Integration (UDDI), an XML-based worldwide business registry.
- WSDL is the language that UDDI uses.
- WSDL is pronounced as 'wiz-dull' and spelled out as 'W-S-D-L'.

WSDL Usage

WSDL is often used in combination with SOAP and XML Schema to provide web services over the Internet. A client program connecting to a web service can read the WSDL to determine what functions are available on the server. Any special datatypes used are embedded in the WSDL file in the form of XML Schema. The client can then use SOAP to actually call one of the functions listed in the WSDL.

History of WSDL

WSDL 1.1 was submitted as a W3C Note by Ariba, IBM, and Microsoft for describing services for the W3C XML Activity on XML Protocols in March 2001.

WSDL 1.1 has not been endorsed by the World Wide Web Consortium (W3C), however it has just released a draft for version 2.0 that will be a recommendation (an official standard), and thus endorsed by the W3C.

Java Web Services:

Different books and different organizations provide different definitions to Web Services. Some of them are listed here.

- A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system. XML is used to encode all communications to a web service. For example, a client invokes a web service by sending an XML message, then waits for a corresponding XML response. As all communication is in XML, web services are not tied to any one operating system or programming language—Java can talk with Perl; Windows applications can talk with Unix applications.

- Web services are self-contained, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains. These applications can be local, distributed, or web-based. Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML.
- Web services are XML-based information exchange systems that use the Internet for direct application-to-application interaction. These systems can include programs, objects, messages, or documents.
- A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

To summarize, a complete web service is, therefore, any service that −

- Is available over the Internet or private (intranet) networks
- Uses a standardized XML messaging system
- Is not tied to any one operating system or programming language
- Is self-describing via a common XML grammar
- Is discoverable via a simple find mechanism

Components of Web Services

The basic web services platform is XML + HTTP. All the standard web services work using the following components −

- SOAP (Simple Object Access Protocol)
- UDDI (Universal Description, Discovery and Integration)
- WSDL (Web Services Description Language)

All these components have been discussed in the Web Services Architecture chapter.

How Does a Web Service Work?

A web service enables communication among various applications by using open standards such as HTML, XML, WSDL, and SOAP. A web service takes the help of −

- XML to tag the data
- SOAP to transfer a message
- WSDL to describe the availability of service.

You can build a Java-based web service on Solaris that is accessible from your Visual Basic program that runs on Windows.

You can also use C# to build new web services on Windows that can be invoked from your web application that is based on JavaServer Pages (JSP) and runs on Linux.

Example

Consider a simple account-management and order processing system. The accounting personnel use a client application built with Visual Basic or JSP to create new accounts and enter new customer orders.

The processing logic for this system is written in Java and resides on a Solaris machine, which also interacts with a database to store information.

The steps to perform this operation are as follows −

- The client program bundles the account registration information into a SOAP message.
- This SOAP message is sent to the web service as the body of an HTTP POST request.
- The web service unpacks the SOAP request and converts it into a command that the application can understand.
- The application processes the information as required and responds with a new unique account number for that customer.
- Next, the web service packages the response into another SOAP message, which it sends back to the client program in response to its HTTP request.
- The client program unpacks the SOAP message to obtain the results of the account registration process.

Web Resources:

Different books and different organizations provide different definitions to Web Services. Some of them are listed here.

- A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system. XML is used to encode all communications to a web service. For example, a client invokes a web service by sending an XML message, then waits for a corresponding XML response. As all communication is in XML, web services are not tied to any one operating system or programming language—Java can talk with Perl; Windows applications can talk with Unix applications.
- Web services are self-contained, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains. These applications can be local, distributed, or web-based. Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML.
- Web services are XML-based information exchange systems that use the Internet for direct application-to-application interaction. These systems can include programs, objects, messages, or documents.
- A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

To summarize, a complete web service is, therefore, any service that −

- Is available over the Internet or private (intranet) networks
- Uses a standardized XML messaging system
- Is not tied to any one operating system or programming language
- Is self-describing via a common XML grammar
- Is discoverable via a simple find mechanism

Components of Web Services

The basic web services platform is XML + HTTP. All the standard web services work using the following components −

- SOAP (Simple Object Access Protocol)
- UDDI (Universal Description, Discovery and Integration)
- WSDL (Web Services Description Language)

All these components have been discussed in the Web Services Architecture chapter.

How Does a Web Service Work?

A web service enables communication among various applications by using open standards such as HTML, XML, WSDL, and SOAP. A web service takes the help of −

- XML to tag the data
- SOAP to transfer a message
- WSDL to describe the availability of service.

You can build a Java-based web service on Solaris that is accessible from your Visual Basic program that runs on Windows.

You can also use C# to build new web services on Windows that can be invoked from your web application that is based on JavaServer Pages (JSP) and runs on Linux.

Example

Consider a simple account-management and order processing system. The accounting personnel use a client application built with Visual Basic or JSP to create new accounts and enter new customer orders.

The processing logic for this system is written in Java and resides on a Solaris machine, which also interacts with a database to store information.

The steps to perform this operation are as follows −

- The client program bundles the account registration information into a SOAP message.
- This SOAP message is sent to the web service as the body of an HTTP POST request.
- The web service unpacks the SOAP request and converts it into a command that the application can understand.
- The application processes the information as required and responds with a new unique account number for that customer.
- Next, the web service packages the response into another SOAP message, which it sends back to the client program in response to its HTTP request.
- The client program unpacks the SOAP message to obtain the results of the account registration process.