# The Browser Object Model:

A Web page is made dynamic by applying JavaScript processing to the HTML elements on that page. Up to this point you probably have considered HTML tags simply as markup codes providing structure to page content and supplying mechanisms through which styling is applied to that content. Importantly, though, HTML tags are also **software objects**. That is, all HTML tags have properties and methods that can be programmed. As is the case with all software objects, **properties**refer to structural, visual, or content characteristics of the element; methods refer to actions the object can perform. HTML tags, then, are programmable through JavaScript processing routines, or scripts, that set their properties and activate their methods in order to make Web pages **dynamic**.

The browser object model (BOM) is a hierarchy of browser objects that are used to manipulate methods and properties associated with the Web browser itself. Objects that make up the BOM include the window object, navigator object, screen object, history, location object, and the document object. The Document Object consists of objects that are used to manipulate methods and properties of the document or Web page loaded in the browser window. The document object represents the Web page currently loaded in the browser window. Each HTML element or tag that makes up the document is also considered an object. It is not necessary to explicitly create any of the objects that make up the browser object model. The objects are automatically created when a Web browser opens a Web page.

# HTML5 :

HTML is the standard markup language for Web pages.

With HTML you can create your own Website.

HTML is easy to learn - You will enjoy it!

### 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>

**HTML5**

HTML5 is the fifth release of HyperText Markup Language. It is used to design the structured content of a web page. In short, HTML5 is one of the several programming languages written to allow for the rendering of a web page. It combines the feature of its predecessor in addition to XHTML and DOM level HTML. Through HTML5 we can design better and user-friendly websites.

Some features of HTML5 are:-

* Header and footer options are more user-friendly now.
* Several APIs allows audio and video option.
* Browser Compatibility-It supports latest versions of most used browsers.
* Extensive multimedia support.

CSS3:

CSS3 is the third and latest release. It has some petty advanced styling capabilities which enable the web designer to manipulate background as well as fonts styles. Web page built-in CSS3 can be easily made responsive using media queries.

Some features of CSS3 are:-

* CSS3 is backward compatible with CSS1.
* CSS3 is the latest iteration of CSS.
* Rounded border feature is available.
* Border image feature is available.
* Text shadow feature is available.
* CSS3 can be split into the module.

HTML5 Canvas

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.

## 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.

XML Namespaces:

XML Namespaces provide a method to avoid element name conflictName Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

This XML carries HTML table information:

<table>  
  <tr>  
   <td>Apples</td>  
    <td>Bananas</td>  
  </tr>  
</table>

This XML carries information about a table (a piece of furniture):

<table>  
  <name>African Coffee Table</name>  
  <width>80</width>  
  <length>120</length>  
</table>

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

A user or an XML application will not know how to handle these differences.

## XML Namespaces - The xmlns Attribute

When using prefixes in XML, a **namespace** for the prefix must be defined.

The namespace can be defined by an **xmlns** attribute in the start tag of an element.

The namespace declaration has the following syntax. xmlns:*prefix*="*URI*".

<root>  
<h:table xmlns:h="http://www.w3.org/TR/html4/">  
  <h:tr>  
    <h:td>Apples</h:td>  
    <h:td>Bananas</h:td>  
  </h:tr>  
</h:table>  
<f:table xmlns:f="https://www.w3schools.com/furniture">  
  <f:name>African Coffee Table</f:name>  
  <f:width>80</f:width>  
  <f:length>120</f:length>  
</f:table>  
</root>

In the example above:

The xmlns attribute in the first <table> element gives the h: prefix a qualified namespace.

The xmlns attribute in the second <table> element gives the f: prefix a qualified namespace.

When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

Namespaces can also be declared in the XML root element:

<root xmlns:h="http://www.w3.org/TR/html4/"  
xmlns:f="https://www.w3schools.com/furniture">  
<h:table>  
  <h:tr>  
    <h:td>Apples</h:td>  
    <h:td>Bananas</h:td>  
  </h:tr>  
</h:table>  
<f:table>  
  <f:name>African Coffee Table</f:name>  
  <f:width>80</f:width>  
  <f:length>120</f:length>  
</f:table>  
</root>

**Note:** The namespace URI is not used by the parser to look up information.

The purpose of using an URI is to give the namespace a unique name.

However, companies often use the namespace as a pointer to a web page containing namespace information.

# XML DTD

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

## What is a DTD?

DTD stands for Document Type Definition.

A DTD defines the structure and the legal elements and attributes of an XML document.

## Valid XML Documents

A "Valid" XML document is "Well Formed", as well as it conforms to the rules of a DTD:

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE note SYSTEM "Note.dtd">  
<note>  
<to>Tove</to>  
<from>Jani</from>  
<heading>Reminder</heading>  
<body>Don't forget me this weekend!</body>  
</note>

The DOCTYPE declaration above contains a reference to a DTD file. The content of the DTD file is shown and explained below.

## XML DTD

The purpose of a DTD is to define the structure and the legal elements and attributes of an XML document:

### Note.dtd:

<!DOCTYPE note  
[  
<!ELEMENT note (to,from,heading,body)>  
<!ELEMENT to (#PCDATA)>  
<!ELEMENT from (#PCDATA)>  
<!ELEMENT heading (#PCDATA)>  
<!ELEMENT body (#PCDATA)>  
]>

The DTD above is interpreted like this:

* !DOCTYPE note -  Defines that the root element of the document is note
* !ELEMENT note - Defines that the note element must contain the elements: "to, from, heading, body"
* !ELEMENT to - Defines the to element to be of type "#PCDATA"
* !ELEMENT from - Defines the from element to be of type "#PCDATA"
* !ELEMENT heading  - Defines the heading element to be of type "#PCDATA"
* !ELEMENT body - Defines the body element to be of type "#PC

## Using DTD for Entity Declaration

A DOCTYPE declaration can also be used to define special characters or strings, used in the document:

### Example

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE note [  
<!ENTITY nbsp "&#xA0;">  
<!ENTITY writer "Writer: Donald Duck.">  
<!ENTITY copyright "Copyright: W3Schools.">  
]>  
<note>  
<to>Tove</to>  
<from>Jani</from>  
<heading>Reminder</heading>  
<body>Don't forget me this weekend!</body>  
<footer>&writer;&nbsp;&copyright;</footer>  
</note>

# XML Schema

An XML Schema describes the structure of an XML document, just like a DTD.

An XML document with correct syntax is called "Well Formed".

An XML document validated against an XML Schema is both "Well Formed" and "Valid".

## XML Schema

XML Schema is an XML-based alternative to DTD:

<xs:element name="note">  
  
<xs:complexType>  
  <xs:sequence>  
    <xs:element name="to" type="xs:string"/>  
    <xs:element name="from" type="xs:string"/>  
    <xs:element name="heading" type="xs:string"/>  
    <xs:element name="body" type="xs:string"/>  
  </xs:sequence>  
</xs:complexType>  
  
</xs:element>

The Schema above is interpreted like this:

* <xs:element name="note"> defines the element called "note"
* <xs:complexType> the "note" element is a complex type
* <xs:sequence> the complex type is a sequence of elements
* <xs:element name="to" type="xs:string"> the element "to" is of type string (text)
* <xs:element name="from" type="xs:string"> the element "from" is of type string
* <xs:element name="heading" type="xs:string"> the element "heading" is of type string
* <xs:element name="body" type="xs:string"> the element "body" is of type string

## XML Schemas are More Powerful than DTD

* XML Schemas are written in XML
* XML Schemas are extensible to additions
* XML Schemas support data types
* XML Schemas support namespaces

## Why Use an XML Schema?

With XML Schema, your XML files can carry a description of its own format.

With XML Schema, independent groups of people can agree on a standard for interchanging data.

With XML Schema, you can verify data.

## XML Schemas Support Data Types

One of the greatest strengths of XML Schemas is the support for data types:

* It is easier to describe document content
* It is easier to define restrictions on data
* It is easier to validate the correctness of data
* It is easier to convert data between different data types

## XML Schemas use XML Syntax

Another great strength about XML Schemas is that they are written in XML:

* You don't have to learn a new language
* You can use your XML editor to edit your Schema files
* You can use your XML parser to parse your Schema files
* You can manipulate your Schemas with the XML DOM
* You can transform your Schemas with XSLT

# Displaying XML

Raw XML files can be viewed in all major browsers.

Don't expect XML files to be displayed as HTML pages.

## Viewing XML Files

<?xml version="1.0" encoding="UTF-8"?>  
 - <note>  
       <to>**Tove**</to>  
       <from>**Jani**</from>  
       <heading>**Reminder**</heading>  
       <body>**Don't forget me this weekend!**</body>  
   </note>

Look at the XML file above in your browser: [note.xml](https://www.w3schools.com/xml/note.xml)

Most browsers will display an XML document with color-coded elements.

Often a plus (+) or minus sign (-) to the left of the elements can be clicked to expand or collapse the element structure.

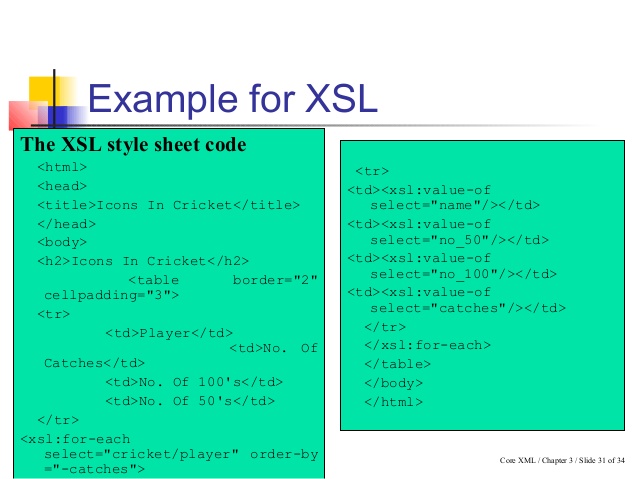
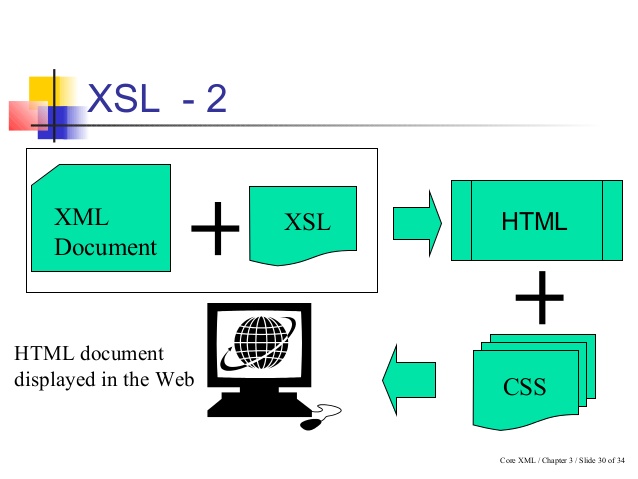
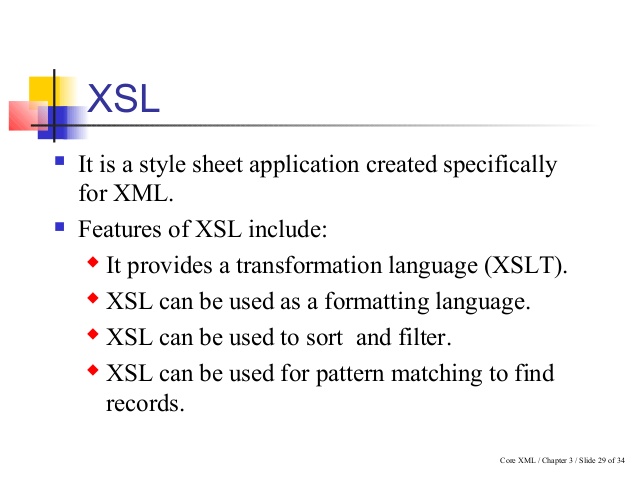
To view raw XML source, try to select "View Page Source" or "View Source" from the browser menu.

**Note:** In Safari 5 (and earlier), only the element text will be displayed. To view the raw XML, you must right click the page and select "View Source".

## Viewing an Invalid XML File

If an erroneous XML file is opened, some browsers will report the error, and some will display it, or display it incorrectly.

<?xml version="1.0" encoding="UTF-8"?>  
 - <note>  
       <to>**Tove**</to>  
       <from>**Jani**</Ffrom>  
       <heading>**Reminder**</heading>  
       <body>**Don't forget me this weekend!**</body>  
   </note>

Try to open the following XML file: [note\_error.xml](javascript:void(0))

**UNIT-II PHP:**

PHP started out as a small open source project that evolved as more and more people found out how useful it was. Rasmus Lerdorf unleashed the first version of PHP way back in 1994. PHP is a recursive acronym for "PHP: Hypertext Preprocessor".

• PHP is a server side scripting language that is embedded in HTML. It is used to manage

• dynamic content, databases, session tracking, even build entire e-commerce sites. It is integrated with a number of popular databases, including MySQL, PostgreSQL,

• Oracle, Sybase, Informix, and Microsoft SQL Server. PHP is pleasingly zippy in its execution, especially when compiled as an Apache module

• on the Unix side. The MySQL server, once started, executes even very complex queries with huge result sets in record-setting time. PHP supports a large number of major protocols such as POP3, IMAP, and LDAP. PHP4

• added support for Java and distributed object architectures (COM and CORBA), making n-tier development a possibility for the first time. PHP is forgiving: PHP language tries to be as forgiving as possible.

• PHP Syntax is C-Like.

**Common Uses of PHP:**

PHP performs system functions, i.e. from files on a system it can create, open, read, write, and close them. The other uses of PHP are: PHP can handle forms, i.e. gather data from files, save data to a file, thru email you

• can send data, return data to the user. You add, delete, modify elements within your database thru PHP.

• Access cookies variables and set cookies.

• Using PHP, you can restrict users to access some pages of your website.

• It can encrypt data.

**Characteristics of PHP:**

Five important characteristics make PHP's practical nature possible:

1)Simplicity

2)Efficiency

3) Security

4)Flexibility

5)Familiarity

**"Hello World" Script in PHP:**

<html>

<head>

<title>Hello World</title>

</head>

<body>

<?php echo "Hello, World!";?>

</body>

</html>

It will produce following result −

Hello, World!

**Installing php:**

In order to develop and run PHP Web pages three vital components need to be installed on your computer system.

* **Web Server** − PHP will work with virtually all Web Server software, including Microsoft's Internet Information Server (IIS) but then most often used is freely available Apache Server. Download Apache for free here − <https://httpd.apache.org/download.cgi>
* **Database** − PHP will work with virtually all database software, including Oracle and Sybase but most commonly used is freely available MySQL database. Download MySQL for free here − <https://www.mysql.com/downloads/>
* **PHP Parser** − In order to process PHP script instructions a parser must be installed to generate HTML output that can be sent to the Web Browser. This tutorial will guide you how to install PHP parser on your computer.

## PHP Parser Installation

Before you proceed it is important to make sure that you have proper environment setup on your machine to develop your web programs using PHP.

Type the following address into your browser's address box.

http://127.0.0.1/info.php

If this displays a page showing your PHP installation related information then it means you have PHP and Webserver installed properly. Otherwise you have to follow given procedure to install PHP on your computer.

This section will guide you to install and configure PHP over the following four platforms −

* [PHP Installation on Linux or Unix with Apache](http://www.tutorialspoint.com/php/php_installation_linux.htm)
* [PHP Installation on Mac OS X with Apache](http://www.tutorialspoint.com/php/php_installation_mac.htm)
* [PHP Installation on Windows NT/2000/XP with IIS](http://www.tutorialspoint.com/php/php_installation_windows_iis.htm)
* [PHP Installation on Windows NT/2000/XP with Apache](http://www.tutorialspoint.com/php/php_installation_windows_apache.htm)

PHP Syntax:

## Escaping to PHP

The PHP parsing engine needs a way to differentiate PHP code from other elements in the page. The mechanism for doing so is known as 'escaping to PHP'. There are four ways to do this −

### Canonical PHP tags

The most universally effective PHP tag style is −

<?php...?>

If you use this style, you can be positive that your tags will always be correctly interpreted.

### Short-open (SGML-style) tags

Short or short-open tags look like this −

<?...?>

Short tags are, as one might expect, the shortest option You must do one of two things to enable PHP to recognize the tags −

* Choose the --enable-short-tags configuration option when you're building PHP.
* Set the short\_open\_tag setting in your php.ini file to on. This option must be disabled to parse XML with PHP because the same syntax is used for XML tags.

### ASP-style tags

ASP-style tags mimic the tags used by Active Server Pages to delineate code blocks. ASP-style tags look like this −

<%...%>

To use ASP-style tags, you will need to set the configuration option in your php.ini file.

### HTML script tags

HTML script tags look like this −

<script language = "PHP">...</script>

## Commenting PHP Code

A *comment* is the portion of a program that exists only for the human reader and stripped out before displaying the programs result. There are two commenting formats in PHP −

**Single-line comments** − They are generally used for short explanations or notes relevant to the local code. Here are the examples of single line comments.

<?

# This is a comment, and

# This is the second line of the comment

// This is a comment too. Each style comments only

print "An example with single line comments";

?>

**Multi-lines printing** − Here are the examples to print multiple lines in a single print statement −

<?

# First Example

print <<<END

This uses the "here document" syntax to output

multiple lines with $variable interpolation. Note

that the here document terminator must appear on a

line with just a semicolon no extra whitespace!

END;

# Second Example

print "This spans

multiple lines. The newlines will be

output as well";

?>

**Multi-lines comments** − They are generally used to provide pseudocode algorithms and more detailed explanations when necessary. The multiline style of commenting is the same as in C. Here are the example of multi lines comments.

<?

/\* This is a comment with multiline

Author : Mohammad Mohtashim

Purpose: Multiline Comments Demo

Subject: PHP

\*/

print "An example with multi line comments";

?>

## PHP is whitespace insensitive

Whitespace is the stuff you type that is typically invisible on the screen, including spaces, tabs, and carriage returns (end-of-line characters).

PHP whitespace insensitive means that it almost never matters how many whitespace characters you have in a row.one whitespace character is the same as many such characters.

For example, each of the following PHP statements that assigns the sum of 2 + 2 to the variable $four is equivalent −

$four = 2 + 2; // single spaces

$four <tab>=<tab2<tab>+<tab>2 ; // spaces and tabs

$four =

2+

2; // multiple lines

## PHP is case sensitive

Yeah it is true that PHP is a case sensitive language. Try out following example −

<html>

<body>

<?php

$capital = 67;

print("Variable capital is $capital<br>");

print("Variable CaPiTaL is $CaPiTaL<br>");

?>

</body>

</html>

This will produce the following result −

Variable capital is 67

Variable CaPiTaL is

## Statements are expressions terminated by semicolons

A *statement* in PHP is any expression that is followed by a semicolon (;).Any sequence of valid PHP statements that is enclosed by the PHP tags is a valid PHP program. Here is a typical statement in PHP, which in this case assigns a string of characters to a variable called $greeting −

$greeting = "Welcome to PHP!";

## Expressions are combinations of tokens

The smallest building blocks of PHP are the indivisible tokens, such as numbers (3.14159), strings (.two.), variables ($two), constants (TRUE), and the special words that make up the syntax of PHP itself like if, else, while, for and so forth

## Braces make blocks

Although statements cannot be combined like expressions, you can always put a sequence of statements anywhere a statement can go by enclosing them in a set of curly braces.

Here both statements are equivalent −

if (3 == 2 + 1)

print("Good - I haven't totally lost my mind.<br>");

if (3 == 2 + 1) {

print("Good - I haven't totally");

print("lost my mind.<br>");

}

## Running PHP Script from Command Prompt

Yes you can run your PHP script on your command prompt. Assuming you have following content in test.php file

<?php

echo "Hello PHP!!!!!";

?>

Now run this script as command prompt as follows −

$ php test.php

It will produce the following result −

Hello PHP!!!!!

**PHP VARIABLES:**

The main way to store information in the middle of a PHP program is by using a variable.

Here are the most important things to know about variables in PHP.

* All variables in PHP are denoted with a leading dollar sign ($).
* The value of a variable is the value of its most recent assignment.
* Variables are assigned with the = operator, with the variable on the left-hand side and the expression to be evaluated on the right.
* Variables can, but do not need, to be declared before assignment.
* Variables in PHP do not have intrinsic types - a variable does not know in advance whether it will be used to store a number or a string of characters.
* Variables used before they are assigned have default values.
* PHP does a good job of automatically converting types from one to another when necessary.
* PHP variables are Perl-like.

PHP has a total of eight data types which we use to construct our variables −

* **Integers** − are whole numbers, without a decimal point, like 4195.
* **Doubles** − are floating-point numbers, like 3.14159 or 49.1.
* **Booleans** − have only two possible values either true or false.
* **NULL** − is a special type that only has one value: NULL.
* **Strings** − are sequences of characters, like 'PHP supports string operations.'
* **Arrays** − are named and indexed collections of other values.
* **Objects** − are instances of programmer-defined classes, which can package up both other kinds of values and functions that are specific to the class.
* **Resources** − are special variables that hold references to resources external to PHP (such as database connections).

The first five are *simple types*, and the next two (arrays and objects) are compound - the compound types can package up other arbitrary values of arbitrary type, whereas the simple types cannot.

We will explain only simple data type in this chapters. Array and Objects will be explained separately.

**PHP ARRAYS:**

An array is a data structure that stores one or more similar type of values in a single value. For example if you want to store 100 numbers then instead of defining 100 variables its easy to define an array of 100 length.

There are three different kind of arrays and each array value is accessed using an ID c which is called array index.

* **Numeric array** − An array with a numeric index. Values are stored and accessed in linear fashion.
* **Associative array** − An array with strings as index. This stores element values in association with key values rather than in a strict linear index order.
* **Multidimensional array** − An array containing one or more arrays and values are accessed using multiple indices

**NOTE** − Built-in array functions is given in function reference [PHP Array Functions](http://www.tutorialspoint.com/php/php_array_functions.htm)

## Numeric Array

These arrays can store numbers, strings and any object but their index will be represented by numbers. By default array index starts from zero.

### Example

Following is the example showing how to create and access numeric arrays.

Here we have used **array()** function to create array. This function is explained in function reference.

<html>

<body>

<?php

/\* First method to create array. \*/

$numbers = array( 1, 2, 3, 4, 5);

foreach( $numbers as $value ) {

echo "Value is $value <br />";

}

/\* Second method to create array. \*/

$numbers[0] = "one";

$numbers[1] = "two";

$numbers[2] = "three";

$numbers[3] = "four";

$numbers[4] = "five";

foreach( $numbers as $value ) {

echo "Value is $value <br />";

}

?>

</body>

</html>

This will produce the following result −

Value is 1

Value is 2

Value is 3

Value is 4

Value is 5

Value is one

Value is two

Value is three

Value is four

Value is five

## Associative Arrays

The associative arrays are very similar to numeric arrays in term of functionality but they are different in terms of their index. Associative array will have their index as string so that you can establish a strong association between key and values.

To store the salaries of employees in an array, a numerically indexed array would not be the best choice. Instead, we could use the employees names as the keys in our associative array, and the value would be their respective salary.

**NOTE** − Don't keep associative array inside double quote while printing otherwise it would not return any value.

### Example

<html>

<body>

<?php

/\* First method to associate create array. \*/

$salaries = array("mohammad" => 2000, "qadir" => 1000, "zara" => 500);

echo "Salary of mohammad is ". $salaries['mohammad'] . "<br />";

echo "Salary of qadir is ". $salaries['qadir']. "<br />";

echo "Salary of zara is ". $salaries['zara']. "<br />";

/\* Second method to create array. \*/

$salaries['mohammad'] = "high";

$salaries['qadir'] = "medium";

$salaries['zara'] = "low";

echo "Salary of mohammad is ". $salaries['mohammad'] . "<br />";

echo "Salary of qadir is ". $salaries['qadir']. "<br />";

echo "Salary of zara is ". $salaries['zara']. "<br />";

?>

</body>

</html>

This will produce the following result −

Salary of mohammad is 2000

Salary of qadir is 1000

Salary of zara is 500

Salary of mohammad is high

Salary of qadir is medium

Salary of zara is low

## Multidimensional Arrays

A multi-dimensional array each element in the main array can also be an array. And each element in the sub-array can be an array, and so on. Values in the multi-dimensional array are accessed using multiple index.

### Example

In this example we create a two dimensional array to store marks of three students in three subjects −

This example is an associative array, you can create numeric array in the same fashion

<html>

<body>

<?php

$marks = array(

"mohammad" => array (

"physics" => 35,

"maths" => 30,

"chemistry" => 39

),

"qadir" => array (

"physics" => 30,

"maths" => 32,

"chemistry" => 29

),

"zara" => array (

"physics" => 31,

"maths" => 22,

"chemistry" => 39

)

);

/\* Accessing multi-dimensional array values \*/

echo "Marks for mohammad in physics : " ;

echo $marks['mohammad']['physics'] . "<br />";

echo "Marks for qadir in maths : ";

echo $marks['qadir']['maths'] . "<br />";

echo "Marks for zara in chemistry : " ;

echo $marks['zara']['chemistry'] . "<br />";

?>

</body>

</html>

This will produce the following result −

Marks for mohammad in physics : 35

Marks for qadir in maths : 32

Marks for zara in chemistry : 39

**PHP STRINGS:**

They are sequences of characters, like "PHP supports string operations".

**NOTE** − Built-in string functions is given in function reference [PHP String Functions](http://in.php.net/manual/en/ref.strings.php)

Following are valid examples of string

$string\_1 = "This is a string in double quotes";

$string\_2 = "This is a somewhat longer, singly quoted string";

$string\_39 = "This string has thirty-nine characters";

$string\_0 = ""; // a string with zero characters

Singly quoted strings are treated almost literally, whereas doubly quoted strings replace variables with their values as well as specially interpreting certain character sequences.

<?php

$variable = "name";

$literally = 'My $variable will not print!\\n';

print($literally);

print "<br />";

$literally = "My $variable will print!\\n";

print($literally);

?>

This will produce the following result −

My $variable will not print!\n

My name will print!\n

There are no artificial limits on string length - within the bounds of available memory, you ought to be able to make arbitrarily long strings.

Strings that are delimited by double quotes (as in "this") are preprocessed in both the following two ways by PHP −

* Certain character sequences beginning with backslash (\) are replaced with special characters
* Variable names (starting with $) are replaced with string representations of their values.

The escape-sequence replacements are −

* \n is replaced by the newline character
* \r is replaced by the carriage-return character
* \t is replaced by the tab character
* \$ is replaced by the dollar sign itself ($)
* \" is replaced by a single double-quote (")
* \\ is replaced by a single backslash (\)

## String Concatenation Operator

To concatenate two string variables together, use the dot (.) operator −

<?php

$string1="Hello World";

$string2="1234";

echo $string1 . " " . $string2;

?>

This will produce the following result −

Hello World 1234

If we look at the code above you see that we used the concatenation operator two times. This is because we had to insert a third string.

Between the two string variables we added a string with a single character, an empty space, to separate the two variables.

## Using the strlen() function

The strlen() function is used to find the length of a string.

Let's find the length of our string "Hello world!" −

<?php

echo strlen("Hello world!");

?>

This will produce the following result −

12

The length of a string is often used in loops or other functions, when it is important to know when the string ends. (i.e. in a loop, we would want to stop the loop after the last character in the string)

## Using the strpos() function

The strpos() function is used to search for a string or character within a string.

If a match is found in the string, this function will return the position of the first match. If no match is found, it will return FALSE.

Let's see if we can find the string "world" in our string

<?php

echo strpos("Hello world!","world");

?>

This will produce the following result −

6

As you see the position of the string "world" in our string is position 6. The reason that it is 6, and not 7, is that the first position in the string is 0, and not 1.

**PHP INTEGERS:**

## Integers

They are whole numbers, without a decimal point, like 4195. They are the simplest type .they correspond to simple whole numbers, both positive and negative. Integers can be assigned to variables, or they can be used in expressions, like so −

$int\_var = 12345;

$another\_int = -12345 + 12345;

Integer can be in decimal (base 10), octal (base 8), and hexadecimal (base 16) format. Decimal format is the default, octal integers are specified with a leading 0, and hexadecimals have a leading 0x.

For most common platforms, the largest integer is (2\*\*31 . 1) (or 2,147,483,647), and the smallest (most negative) integer is . (2\*\*31 . 1) (or .2,147,483,647).

## Doubles

They like 3.14159 or 49.1. By default, doubles print with the minimum number of decimal places needed. For example, the code −

<?php

$many = 2.2888800;

$many\_2 = 2.2111200;

$few = $many + $many\_2;

print("$many + $many\_2 = $few <br>");

?>

It produces the following browser output −

2.28888 + 2.21112 = 4.5

**RUBY:**

Ruby is −

* A high-level programming language.
* Interpreted like Perl, Python, Tcl/TK.
* Object-oriented like Smalltalk, Eiffel, Ada, Java.

## Why Ruby?

Ruby originated in Japan and now it is gaining popularity in US and Europe as well. The following factors contribute towards its popularity −

* Easy to learn
* Open source (very liberal license)
* Rich libraries
* Very easy to extend
* Truly object-oriented
* Less coding with fewer bugs
* Helpful community

Although we have many reasons to use Ruby, there are a few drawbacks as well that you may have to consider before implementing Ruby −

* **Performance Issues** − Although it rivals Perl and Python, it is still an interpreted language and we cannot compare it with high-level programming languages like C or C++.
* **Threading model** − Ruby does not use native threads. Ruby threads are simulated in the VM rather than running as native OS threads.

## Sample Ruby Code

Here is a sample Ruby code to print "Hello Ruby"

# The Hello Class

class Hello

def initialize( name )

@name = name.capitalize

end

def salute

puts "Hello #{@name}!"

end

end

# Create a new object

h = Hello.new("Ruby")

# Output "Hello Ruby!"

h.salute

**Output** − This will produce the following result –

Hello Ruby!

## Defining a Class in Ruby

To implement object-oriented programming by using Ruby, you need to first learn how to create objects and classes in Ruby.

A class in Ruby always starts with the keyword *class* followed by the name of the class. The name should always be in initial capitals. The class *Customer* can be displayed as −

class Customer

end

You terminate a class by using the keyword *end*. All the data members in the *class* are between the class definition and the *end* keyword.

## Variables in a Ruby Class

Ruby provides four types of variables −

* **Local Variables** − Local variables are the variables that are defined in a method. Local variables are not available outside the method. You will see more details about method in subsequent chapter. Local variables begin with a lowercase letter or \_.
* **Instance Variables** − Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.
* **Class Variables** − Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.
* **Global Variables** − Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($).

### Example

Using the class variable @@no\_of\_customers, you can determine the number of objects that are being created. This enables in deriving the number of customers.

class Customer

@@no\_of\_customers = 0

end

## Creating Objects in Ruby using new Method

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. You can create objects in Ruby by using the method *new* of the class.

The method *new* is a unique type of method, which is predefined in the Ruby library. The new method belongs to the *class* methods.

Here is the example to create two objects cust1 and cust2 of the class Customer −

cust1 = Customer. new

cust2 = Customer. new

Here, cust1 and cust2 are the names of two objects. You write the object name followed by the equal to sign (=) after which the class name will follow. Then, the dot operator and the keyword *new* will follow.

## Custom Method to Create Ruby Objects

You can pass parameters to method *new* and those parameters can be used to initialize class variables.

When you plan to declare the *new* method with parameters, you need to declare the method *initialize* at the time of the class creation.

The *initialize* method is a special type of method, which will be executed when the *new* method of the class is called with parameters.

Here is the example to create initialize method −

class Customer

@@no\_of\_customers = 0

def initialize(id, name, addr)

@cust\_id = id

@cust\_name = name

@cust\_addr = addr

end

end

In this example, you declare the *initialize* method with **id, name**, and **addr** as local variables. Here, *def* and *end* are used to define a Ruby method *initialize*. You will learn more about methods in subsequent chapters.

In the *initialize* method, you pass on the values of these local variables to the instance variables @cust\_id, @cust\_name, and @cust\_addr. Here local variables hold the values that are passed along with the new method.

Now, you can create objects as follows −

cust1 = Customer.new("1", "John", "Wisdom Apartments, Ludhiya")

cust2 = Customer.new("2", "Poul", "New Empire road, Khandala")

## Member Functions in Ruby Class

In Ruby, functions are called methods. Each method in a *class* starts with the keyword *def* followed by the method name.

The method name always preferred in **lowercase letters**. You end a method in Ruby by using the keyword *end*.

Here is the example to define a Ruby method −

class Sample

def function

statement 1

statement 2

end

end

Here, *statement 1* and *statement 2* are part of the body of the method *function* inside the class Sample. These statments could be any valid Ruby statement. For example we can put a method *puts* to print *Hello Ruby* as follows −

class Sample

def hello

puts "Hello Ruby!"

end

end

Now in the following example, create one object of Sample class and call *hello* method and see the result −

#!/usr/bin/ruby

class Sample

def hello

puts "Hello Ruby!"

end

end

# Now using above class to create objects

object = Sample. new

object.hello

This will produce the following result −

Hello Ruby!

**HASH:**

A Hash is a collection of key-value pairs like this: "employee" = > "salary". It is similar to an Array, except that indexing is done via arbitrary keys of any object type, not an integer index.

The order in which you traverse a hash by either key or value may seem arbitrary and will generally not be in the insertion order. If you attempt to access a hash with a key that does not exist, the method will return *nil*.

## Creating Hashes

As with arrays, there is a variety of ways to create hashes. You can create an empty hash with the *new* class method −

months = Hash.new

You can also use *new* to create a hash with a default value, which is otherwise just *nil* −

months = Hash.new( "month" )

or

months = Hash.new "month"

When you access any key in a hash that has a default value, if the key or value doesn't exist, accessing the hash will return the default value −

#!/usr/bin/ruby

months = Hash.new( "month" )

puts "#{months[0]}"

puts "#{months[72]}"

This will produce the following result −

Month

month

#!/usr/bin/ruby

H = Hash["a" => 100, "b" => 200]

puts "#{H['a']}"

puts "#{H['b']}"

This will produce the following result −

100

200

You can use any Ruby object as a key or value, even an array, so the following example is a valid one −

[1,"jan"] => "January"

## Hash Built-in Methods

We need to have an instance of Hash object to call a Hash method. As we have seen, following is the way to create an instance of Hash object −

Hash[[key =>|, value]\* ] or

Hash.new [or] Hash.new(obj) [or]

Hash.new { |hash, key| block }

This will return a new hash populated with the given objects. Now using the created object, we can call any available instance methods. For example −

#!/usr/bin/ruby

$, = ", "

months = Hash.new( "month" )

months = {"1" => "January", "2" => "February"}

keys = months.keys

puts "#{keys}"

This will produce the following result −

["1", "2"]

**RUBY METHODS:**

Ruby methods are very similar to functions in any other programming language. Ruby methods are used to bundle one or more repeatable statements into a single unit.

Method names should begin with a lowercase letter. If you begin a method name with an uppercase letter, Ruby might think that it is a constant and hence can parse the call incorrectly.

Methods should be defined before calling them, otherwise Ruby will raise an exception for undefined method invoking.

## Syntax

def method\_name [( [arg [= default]]...[, \* arg [, &expr ]])]

expr..

end

So, you can define a simple method as follows −

def method\_name

expr..

end

You can represent a method that accepts parameters like this −

def method\_name (var1, var2)

expr..

end

You can set default values for the parameters, which will be used if method is called without passing the required parameters −

def method\_name (var1 = value1, var2 = value2)

expr..

end

Whenever you call the simple method, you write only the method name as follows −

method\_name

However, when you call a method with parameters, you write the method name along with the parameters, such as −

method\_name 25, 30

The most important drawback to using methods with parameters is that you need to remember the number of parameters whenever you call such methods. For example, if a method accepts three parameters and you pass only two, then Ruby displays an error.

### Example

#!/usr/bin/ruby

def test(a1 = "Ruby", a2 = "Perl")

puts "The programming language is #{a1}"

puts "The programming language is #{a2}"

end

test "C", "C++"

test

This will produce the following result −

The programming language is C

The programming language is C++

The programming language is Ruby

The programming language is Perl

## Return Values from Methods

Every method in Ruby returns a value by default. This returned value will be the value of the last statement. For example −

def test

i = 100

j = 10

k = 0

end

This method, when called, will return the last declared variable *k*.

## Ruby return Statement

The *return* statement in ruby is used to return one or more values from a Ruby Method.

### Syntax

return [expr[`,' expr...]]

If more than two expressions are given, the array containing these values will be the return value. If no expression given, nil will be the return value.

### Example

return

OR

return 12

OR

return 1,2,3

Have a look at this example −

#!/usr/bin/ruby

def test

i = 100

j = 200

k = 300

return i, j, k

end

var = test

puts var

This will produce the following result −

100

200

300

## Variable Number of Parameters

Suppose you declare a method that takes two parameters, whenever you call this method, you need to pass two parameters along with it.

However, Ruby allows you to declare methods that work with a variable number of parameters. Let us examine a sample of this −

#!/usr/bin/ruby

def sample (\*test)

puts "The number of parameters is #{test.length}"

for i in 0...test.length

puts "The parameters are #{test[i]}"

end

end

sample "Zara", "6", "F"

sample "Mac", "36", "M", "MCA"

In this code, you have declared a method sample that accepts one parameter test. However, this parameter is a variable parameter. This means that this parameter can take in any number of variables. So, the above code will produce the following result −

The number of parameters is 3

The parameters are Zara

The parameters are 6

The parameters are F

The number of parameters is 4

The parameters are Mac

The parameters are 36

The parameters are M

The parameters are MCA

## Class Methods

When a method is defined outside of the class definition, the method is marked as *private* by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the *private* mark of the methods can be changed by *public* or *private* of the Module.

Whenever you want to access a method of a class, you first need to instantiate the class. Then, using the object, you can access any member of the class.

Ruby gives you a way to access a method without instantiating a class. Let us see how a class method is declared and accessed −

class Accounts

def reading\_charge

end

def Accounts.return\_date

end

end

See how the method return\_date is declared. It is declared with the class name followed by a period, which is followed by the name of the method. You can access this class method directly as follows −

Accounts.return\_date

To access this method, you need not create objects of the class Accounts.

## Ruby alias Statement

This gives alias to methods or global variables. Aliases cannot be defined within the method body. The alias of the method keeps the current definition of the method, even when methods are overridden.

Making aliases for the numbered global variables ($1, $2,...) is prohibited. Overriding the built-in global variables may cause serious problems.

### Syntax

alias method-name method-name

alias global-variable-name global-variable-name

### Example

alias foo bar

alias $MATCH $&

Here we have defined foo alias for bar, and $MATCH is an alias for $&

## Ruby undef Statement

This cancels the method definition. An *undef* cannot appear in the method body.

By using *undef* and *alias*, the interface of the class can be modified independently from the superclass, but notice it may be broke programs by the internal method call to self.

### Syntax

undef method-name

### Example

To undefine a method called *bar* do the following −

undef bar

**RUBY STATEMENTS:**

Ruby offers conditional structures that are pretty common to modern languages. Here, we will explain all the conditional statements and modifiers available in Ruby.

## Ruby if...else Statement

### Syntax

if conditional [then]

code...

[elsif conditional [then]

code...]...

[else

code...]

end

*if* expressions are used for conditional execution. The values *false* and *nil* are false, and everything else are true. Notice Ruby uses elsif, not else if nor elif.

Executes *code* if the *conditional* is true. If the *conditional* is not true, *code* specified in the else clause is executed.

An if expression's *conditional* is separated from code by the reserved word *then*, a newline, or a semicolon.

### Example

#!/usr/bin/ruby

x = 1

if x > 2

puts "x is greater than 2"

elsif x <= 2 and x!=0

puts "x is 1"

else

puts "I can't guess the number"

end

x is 1

## Ruby if modifier

### Syntax

code if condition

Executes *code* if the *conditional* is true.

### Example

#!/usr/bin/ruby

$debug = 1

print "debug\n" if $debug

This will produce the following result −

debug

## Ruby unless Statement

### Syntax

unless conditional [then]

code

[else

code ]

end

Executes *code* if *conditional* is false. If the *conditional* is true, code specified in the else clause is executed.

### example

#!/usr/bin/ruby

x = 1

unless x>=2

puts "x is less than 2"

else

puts "x is greater than 2"

end

This will produce the following result −

x is less than 2

## Ruby unless modifier

### Syntax

code unless conditional

Executes *code* if *conditional* is false.

### Example

#!/usr/bin/ruby

$var = 1

print "1 -- Value is set\n" if $var

print "2 -- Value is set\n" unless $var

$var = false

print "3 -- Value is set\n" unless $var

This will produce the following result −

1 -- Value is set

3 -- Value is set

## Ruby case Statement

### Syntax

case expression

[when expression [, expression ...] [then]

code ]...

[else

code ]

end

Compares the *expression* specified by case and that specified by when using the === operator and executes the *code* of the when clause that matches.

The *expression* specified by the when clause is evaluated as the left operand. If no when clauses match, *case* executes the code of the *else* clause.

A *when* statement's expression is separated from code by the reserved word then, a newline, or a semicolon. Thus −

case expr0

when expr1, expr2

stmt1

when expr3, expr4

stmt2

else

stmt3

end

is basically similar to the following −

\_tmp = expr0

if expr1 === \_tmp || expr2 === \_tmp

stmt1

elsif expr3 === \_tmp || expr4 === \_tmp

stmt2

else

stmt3

end

### Example

#!/usr/bin/ruby

$age = 5

case $age

when 0 .. 2

puts "baby"

when 3 .. 6

puts "little child"

when 7 .. 12

puts "child"

when 13 .. 18

puts "youth"

else

puts "adult"

end

This will produce the following result −

little child

**LOOP:**

Loops in Ruby are used to execute the same block of code a specified number of times. This chapter details all the loop statements supported by Ruby.

## Ruby while Statement

### Syntax

while conditional [do]

code

end

Executes *code* while *conditional* is true. A *while* loop's *conditional* is separated from *code* by the reserved word do, a newline, backslash \, or a semicolon ;.

### Example

#!/usr/bin/ruby

$i = 0

$num = 5

while $i < $num do

puts("Inside the loop i = #$i" )

$i +=1

end

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

## Ruby while modifier

### Syntax

code while condition

OR

begin

code

end while conditional

Executes *code* while *conditional* is true.

If a *while* modifier follows a *begin* statement with no *rescue* or ensure clauses, *code* is executed once before conditional is evaluated.

### Example

#!/usr/bin/ruby

$i = 0

$num = 5

begin

puts("Inside the loop i = #$i" )

$i +=1

end while $i < $num

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

## Ruby until Statement

until conditional [do]

code

end

Executes *code* while *conditional* is false. An *until* statement's conditional is separated from *code* by the reserved word *do*, a newline, or a semicolon.

### Example

#!/usr/bin/ruby

$i = 0

$num = 5

until $i > $num do

puts("Inside the loop i = #$i" )

$i +=1;

end

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

Inside the loop i = 5

## Ruby until modifier

### Syntax

code until conditional

OR

begin

code

end until conditional

Executes *code* while *conditional* is false.

If an *until* modifier follows a *begin* statement with no *rescue* or ensure clauses, *code* is executed once before *conditional* is evaluated.

### Example

#!/usr/bin/ruby

$i = 0

$num = 5

begin

puts("Inside the loop i = #$i" )

$i +=1;

end until $i > $num

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

Inside the loop i = 5

## Ruby for Statement

### Syntax

for variable [, variable ...] in expression [do]

code

end

Executes *code* once for each element in *expression*.

### Example

#!/usr/bin/ruby

for i in 0..5

puts "Value of local variable is #{i}"

end

Here, we have defined the range 0..5. The statement for *i* in 0..5 will allow *i* to take values in the range from 0 to 5 (including 5). This will produce the following result −

Value of local variable is 0

Value of local variable is 1

Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5

A *for...in* loop is almost exactly equivalent to the following −

(expression).each do |variable[, variable...]| code end

except that a *for* loop doesn't create a new scope for local variables. A *for* loop's *expression* is separated from *code* by the reserved word do, a newline, or a semicolon.

### Example

#!/usr/bin/ruby

(0..5).each do |i|

puts "Value of local variable is #{i}"

end

This will produce the following result −

Value of local variable is 0

Value of local variable is 1

Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5

## Ruby break Statement

### Syntax

break

Terminates the most internal loop. Terminates a method with an associated block if called within the block (with the method returning nil).

### Example

#!/usr/bin/ruby

for i in 0..5

if i > 2 then

break

end

puts "Value of local variable is #{i}"

end

This will produce the following result −

Value of local variable is 0

Value of local variable is 1

Value of local variable is 2

## Ruby next Statement

### Syntax

next

Jumps to the next iteration of the most internal loop. Terminates execution of a block if called within a block (with *yield* or call returning nil).

### Example

#!/usr/bin/ruby

for i in 0..5

if i < 2 then

next

end

puts "Value of local variable is #{i}"

end

This will produce the following result −

Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5

## Ruby redo Statement

### Syntax

redo

Restarts this iteration of the most internal loop, without checking loop condition. Restarts *yield* or *call* if called within a block.

### Example

#!/usr/bin/ruby

for i in 0..5

if i < 2 then

puts "Value of local variable is #{i}"

redo

end

end

This will produce the following result and will go in an infinite loop −

Value of local variable is 0

Value of local variable is 0

............................

## Ruby retry Statement

### Syntax

retry

If *retry* appears in rescue clause of begin expression, restart from the beginning of the begin body.

begin

do\_something # exception raised

rescue

# handles error

retry # restart from beginning

end

If retry appears in the iterator, the block, or the body of the *for* expression, restarts the invocation of the iterator call. Arguments to the iterator is re-evaluated.

for i in 1..5

retry if some\_condition # restart from i == 1

end

### Example

#!/usr/bin/ruby

for i in 0..5

retry if i > 2

puts "Value of local variable is #{i}"

end

This will produce the following result and will go in an infinite loop −

Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

............................