DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DIGITAL NOTES

ON

SCRIPTING LANGUAGES

R22A0518

B. TECH III YEAR – II SEM

(R22) REGULATION

(2024-25)

Prepared by
D SAI ESWARI
Asst.Professor

MALLAREDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution–UGC, Govt.of India)
Recognized under2(f)and12(B) of UGC ACT1956
(Affiliated to JNTUH,Hyderabad,ApprovedbyAICTE-AccreditedbyNBA&NAAC–'A'Grade-ISO9001:2015Certified)
Maisammaguda, Dhulapally(PostVia.Hakimpet),Secunderabad–500100,TelanganaState,India

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

To acknowledge quality education and instill high patterns of discipline making the students technologically superior and ethically strong which involves the improvement in the quality of life in human race.

Mission

- To achieve and impart holistic technical education using the best of infrastructure, outstanding technical and teaching expertise to establish the students into competent and confident engineers.
- Evolving the center of excellence through creative and innovative teaching learning practicesforpromotingacademicachievementtoproduceinternationallyacceptedcompetiti veand world class professionals.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1–ANALYTICALSKILLS

- To facilitate the graduates with the ability to visualize, gather information, articulate, analyze, solve complex problems, and make decisions. These are essential to address the challenges of complex and computation intensive problems increasing their productivity.

PEO2–TECHNICALSKILLS

- Tofacilitatethegraduateswiththetechnicalskillsthatpreparethemforimmediateemploymentandpursue certification providing a deeper understanding of the technology in advanced areas of computer science and related fields, thus encouraging pursuing higher education and research based on their interest.

PEO3–SOFTSKILLS

- To facilitate the graduates with the soft skills that include fulfilling the mission, setting goals, showing self confidence by communicating effectively, having a positive attitude, get involved in team-work, being a leader, managing their career and their life.

PEO4–PROFESSIONALETHICS

- To facilitate the graduates with the knowledge of professional and ethical responsibilities by paying attention to grooming, being conservative with style, following dress codes, safety codes, and adapting them to technological advancements.

PROGRAM SPECIFIC OUTCOMES (PSOs)

After the completion of the course, B.Tech Computer Science and Engineering, the graduates will have the following Program Specific Outcomes:

1.FundamentalsandcriticalknowledgeoftheComputerSystem:- AbletoUnderstandtheworkingprinciples of the computer System and its components, Apply the knowledge to build, asses, and analyze the software and hardware aspects of it.

2.The comprehensive and Applicative knowledge of Software Development: Comprehensive skills of Programming Languages, Software process models, methodologies, and able to plan, develop, test, analyze, and manage the software and hardware intensive systems in heterogeneous platforms individually or working in teams.

3.Applications of Computing Domain & Research: Able to use the professional, managerial, interdisciplinary skill set, and domain specific tools in development processes, identify their search gaps, and provide innovative solutions to them.

PROGRAM OUTCOMES (POs)

Engineering Graduates should possess the following:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals,andanengineeringspecializationtothesolutionofcomplexengineeringproblems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. Design / development of solutions: Design solutions for complex engineeringproblemsanddesignsystemcomponentsorprocessesthatmeetthespecifiedneedswithappropriateconsideration for thepublic health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

SYLLABUS

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

III Year B.Tech CSE-II SEM **L/T/P/C**
3/ -/ - /3

(R22A0518) SCRIPTING LANGUAGES

COURSE OBJECTIVES:
This course will enable students to

1. To study the basics of scripting languages like Java script, Perl, PHP and Ruby
2. To understand the requirements of Scripting Languages
3. To identify the uses of Scripting Languages
4. To introduce in-depth knowledge of programming features of Perl and PHP.
5. To state the implementation and applications of Scripting.

UNIT- I

Introduction to Scripts and Scripting Languages – Scripts and Programs, Uses for Scripting Languages, Web Scripting.
JavaScript: Variables, DataTypes, Operators, Conditional statements, Loops, Arrays, Functions, Objects- Predefined objects, Accessing objects, Object Methods.

U NIT- II

JavaScript programming of reactive web pages elements:

JavaScript Events- Mouse events, Keyboard events, Form events, window events, Event handlers, Frames, Form object, JavaScript Form Validation.

UNIT- III

PERL : Data Types, Variables, Scalars, Operators, Conditional statements , Loops, Arrays ,
Strings , Hashes ,
Lists , Built-in Functions, Pattern matching and regular expression operators.

UNIT -IV

PHP : Data Types, Variables, Operators, Conditional statements, Loops ,Arrays - Indexed
Array, Associative
Array, String Functions, Functions- Parameterized Function, Call By Value, all By Reference ,
File Handling,
PHP Form handling.

UNIT- V

Ruby : Data types, Variables, Operators, Conditional statements, Loops, Methods, Blocks, Modules, Arrays, Strings, Hashes, File I/O, Ruby Form handling.

TEXT BOOKS:

1. The World of Scripting Languages, David Barron, Wiley Publications.
2. Learning PHP, MySQL, JavaScript, CSS & HTML5: A Step-by-Step Guide to Creating DynamicWebsites 3rdEdition,O’ReillyPublications

REFERENCE BOOKS:

1. The Ruby Programming Language, David Flanagan and Yukihiro Matsumoto, O’Reilly Publications.
2. Beginning JavaScript with Dom scripting and AJAX, Russ Ferguson, Christian Heilmann, Apress.
3. Programming Perl, Larry Wall, T. Christiansen and J. Orwant, O’Reilly, SPD.
4. Open Source Web Development with LAMP using Linux Apache, MySQL, Perl and PHP, J. Lee and B. Ware (Addison Wesley) Pearson Education.

COURSE OUTCOMES:

The students will be able:

1. Comprehend the differences between typical scripting languages and typical system and application programming languages.
2. To implement the design of programs for simple applications.
3. To write and apply Perl & PHP scripts.
4. Gain knowledge of the strengths and weakness of Perl, and Ruby.
5. To create software systems using scripting languages such as Perl, PHP, and Ruby.

INDEX

UNIT-I:
Introduction to Scripts and Scripting Languages: Scripts and Programs, Uses for Scripting Languages, Web Scripting.
JavaScript: Variables, Data Types, Operators, Conditional statements, Loops, Arrays, Functions, Objects- Predefined objects, Accessing objects, Object Methods.

Scripting Languages

Scripting languages, as the name suggests, is a programming language that supports scripts. A scripting language binds a set of software components that collaborate to solve a particular problem. Scripting assumes the existence of powerful components and provides the means to connect them together. Scripting languages are glue languages that integrate the execution of system utilities including compilers; command line interpretation; shell-based programming; and execution of codes written in web-based languages. The purpose of a scripting language is the development of applications by plugging existing components together and they generally favor high-level programming over execution speed. Scripting is used in a variety of applications, and scripting languages are correspondingly diverse. Python is a powerful scripting language for complex system involving operating system, networks, and web-based programming.

Programming Languages

A programming language is an organized way of communicating with a computer, such that the computer behaves according to the instructions given by the programmer. A programming language is an artificial formalism in which algorithms can be expressed. In the modern era, the problems to be solved by computers lie in different problem domains such as scientific computing, database programming, business applications, process automation, and web-based applications. All these domains are quite different with varied requirements. A programming language is a specific set of instructions given to a computer in a language that the computer understands to perform specific tasks. Today’s programming languages are the product of development that started in the 1950s. The term programming languages usually refer to high-level languages such as C++, Java, Ada, Pascal, and FORTRAN.

Uses for scripting languages

The functions and applications of scripting languages vary based on the type of scripting language you're using. There are many uses for scripting languages, including:

- Task automation: Programmers often use scripting languages to automate task execution within a runtime environment. This involves writing code that allows individuals to use software to complete repetitive, predictable and straightforward tasks, such as paying bills from an account and sending notifications via email.
- Content display for web applications: Programmers use scripting to ensure programs run correctly on the server and display the functional and interactive

content on a webpage, such as images and links.

- Command sequences: Many programmers apply scripting languages to condense command sequences, allowing the program to run faster and improving the functionality of parent applications.
- Data extraction: Programmers use scripting languages to pull data from data sets, such as in data analysis, research and statistics.
- Dynamic web apps: Programmers use a variety of scripting languages to power webpages and applications on the server side with efficient code and clear instructions for displaying dynamic content, which is data that changes based on the user's behavior or preferences.
- System administration: When administrators want to generate and pull data, guide user queries and improve systems, they use scripting languages.
- Game modding: Game modification creators use scripting languages to make custom content for games with unique functionality and designs that improve regular gameplay.

Web Scripting

- The process of creating and embedding scripts in a web page is known as web-scripting. A script or a computer-script is a list of commands that are embedded in a web-page normally and are interpreted and executed by a certain program or scripting engine.
- Scripts may be written for a variety of purposes such as for automating processes on a local-computer or to generate web pages.
- The programming languages in which scripts are written are called scripting language, there are many scripting languages available today.
- Common scripting languages are VBScript, JavaScript, ASP, PHP, PERL, JSP etc.

Java Script

JavaScript (JS) is the world's most popular lightweight, interpreted compiled programming language. It is also known as a scripting language for web pages. It can be used for **Client-side** as well as **Server-side** developments.JavaScript is the most popular and hence the most loved language around the globe. Apart from this, there are abundant reasons to learn it.

Below are a listing of few important points:

- **No need of compilers:** Since JavaScript is an interpreted language, therefore it does not need any compiler for compilations.
- **Used both Client and Server-side:** Earlier JavaScript was used to build client-side applications only, but with the evolution of its frameworks namely Node.js and Express.js, it is now widely used for building server-side applications too.

- **Helps to build a complete solution:** As we saw, JavaScript is widely used in both client and server-side applications, therefore it helps us to build an end-to-end solution to a given problem.
- **Used everywhere:** JavaScript is so loved because it can be used anywhere. It can be used to develop websites, games or mobile apps, etc.
- **Huge community support:** JavaScript has a huge community of users and mentors who love this language and take it's legacy forward.

Variables

Variables are containers for storing data (storing data values).

Declare a JavaScript Variable:

- Using var
- Using let
- Using const

declared with the var keyword:

```
var x=5;
var y=6;
var z=x+y;
o/p: The value of z is: 11
```

In this example, x, y, and z, are variables, declared with the let keyword:

```
let x=5;
let y=6;
let z=x+y;
o/p: The value of z is: 11
```

If you want a general rule: always declare variables with const.

If you think the value of the variable can change, use let.

```
const p1=5;
const p2=6;
Let tot=p1=p2;
o/p: The total is: 11
```

Datatypes

- **Numbers**: Represent both integer and floating-point numbers. Example: 5, 6.5, 7 etc.
- **String**: A string is a sequence of characters. In JavaScript, strings can be enclosed within the single or double quotes. Example: "Hello GeeksforGeeks" etc.
- **Boolean**: Represent a logical entity and can have two values: true or false.
- **Null**: This type has only one value: null. It is left intentionally so that it shows something that does not exist.
- **Undefined**: A variable that has not been assigned a value is undefined.

- **Symbol:** Unlike other primitive data types, it does not have any literal form. It is a built-in object whose constructor returns a symbol-that is unique.
- **bigint:** The bigint type represents the whole numbers that are larger than 2^{53}-1. To form a bigint literal number, you append the letter n at the end of the number.
- **Object**: It is the most important data-type and forms the building blocks for modern JavaScript. We will learn about these data types in detail in further articles.

Operators

JavaScript operators are symbols that are used to perform operations on operands. For example:

1. var sum=10+20;

Here, + is the arithmetic operator and = is the assignment operator.

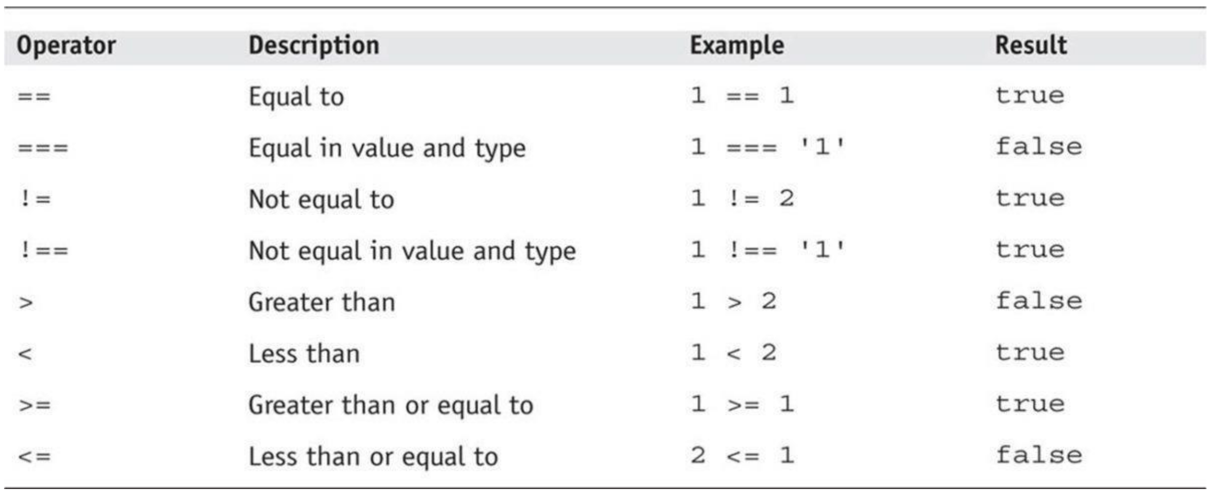
There are following types of operators in JavaScript.

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Bitwise Operators
4. Logical Operators
5. Assignment Operators
6. Special Operators

Arithmetic Operators

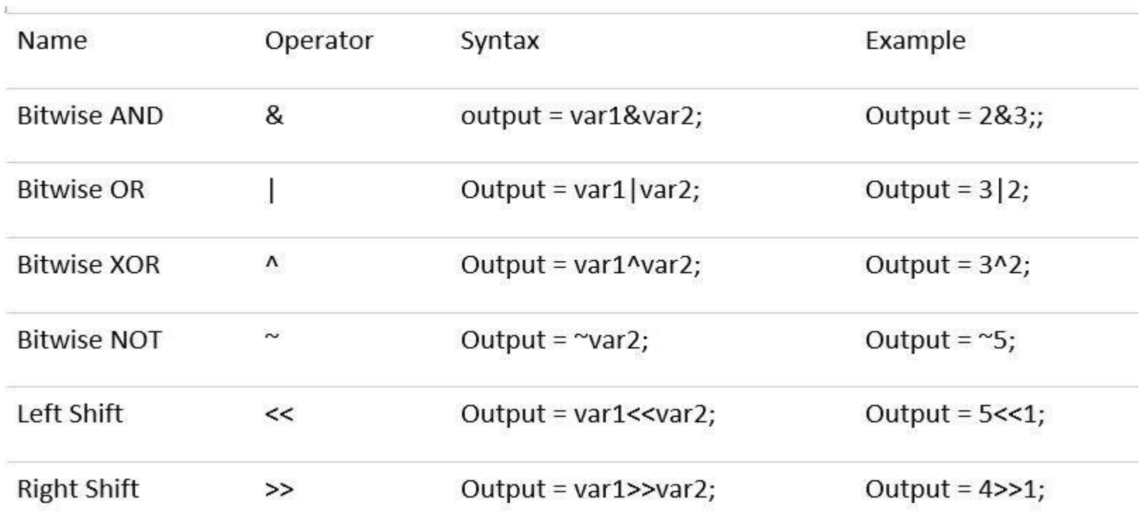
Arithmetic operators are used to perform arithmetic operations on the operands. The following operators are known as JavaScript arithmetic operators.

Comparison operators

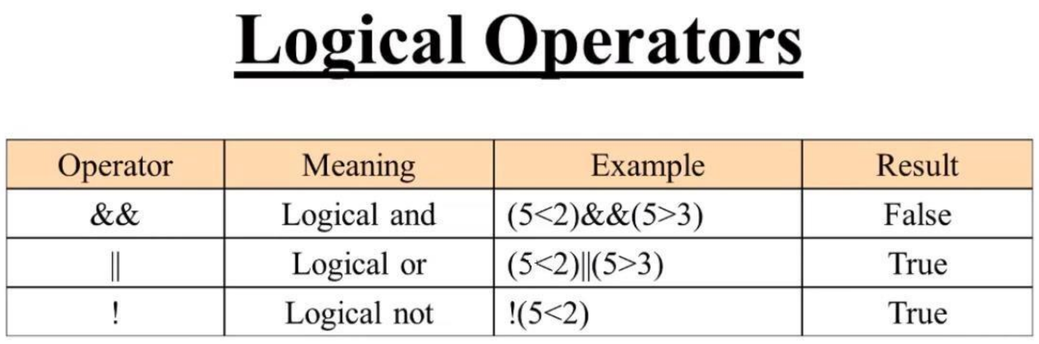


Operator	Description	Example	Result
==	Equal to	`1 == 1`	`true`
===	Equal in value and type	`1 === '1'`	`false`
!=	Not equal to	`1 != 2`	`true`
!==	Not equal in value and type	`1 !== '1'`	`true`
>	Greater than	`1 > 2`	`false`
<	Less than	`1 < 2`	`true`
>=	Greater than or equal to	`1 >= 1`	`true`
<=	Less than or equal to	`2 <= 1`	`false`

Bitwise operators



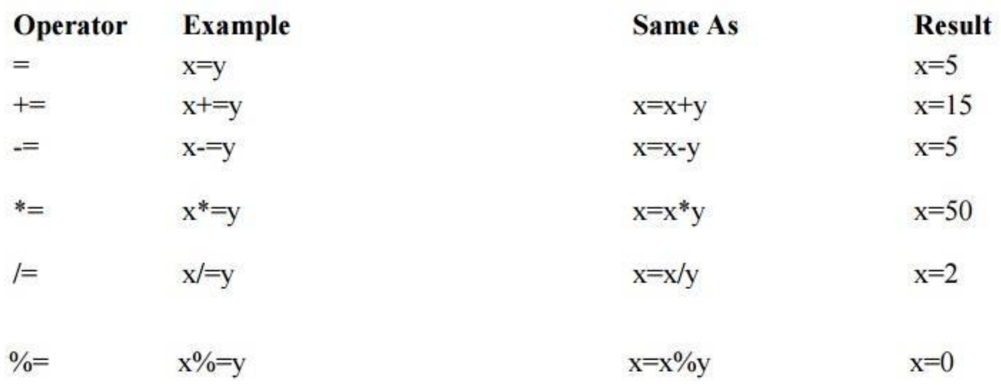
Name	Operator	Syntax	Example
Bitwise AND	&	output = var1&var2;	Output = 2&3;;
Bitwise OR	|	Output = var1|var2;	Output = 3|2;
Bitwise XOR	^	Output = var1^var2;	Output = 3^2;
Bitwise NOT	~	Output = ~var2;	Output = ~5;
Left Shift	<<	Output = var1<<var2;	Output = 5<<1;
Right Shift	>>	Output = var1>>var2;	Output = 4>>1;



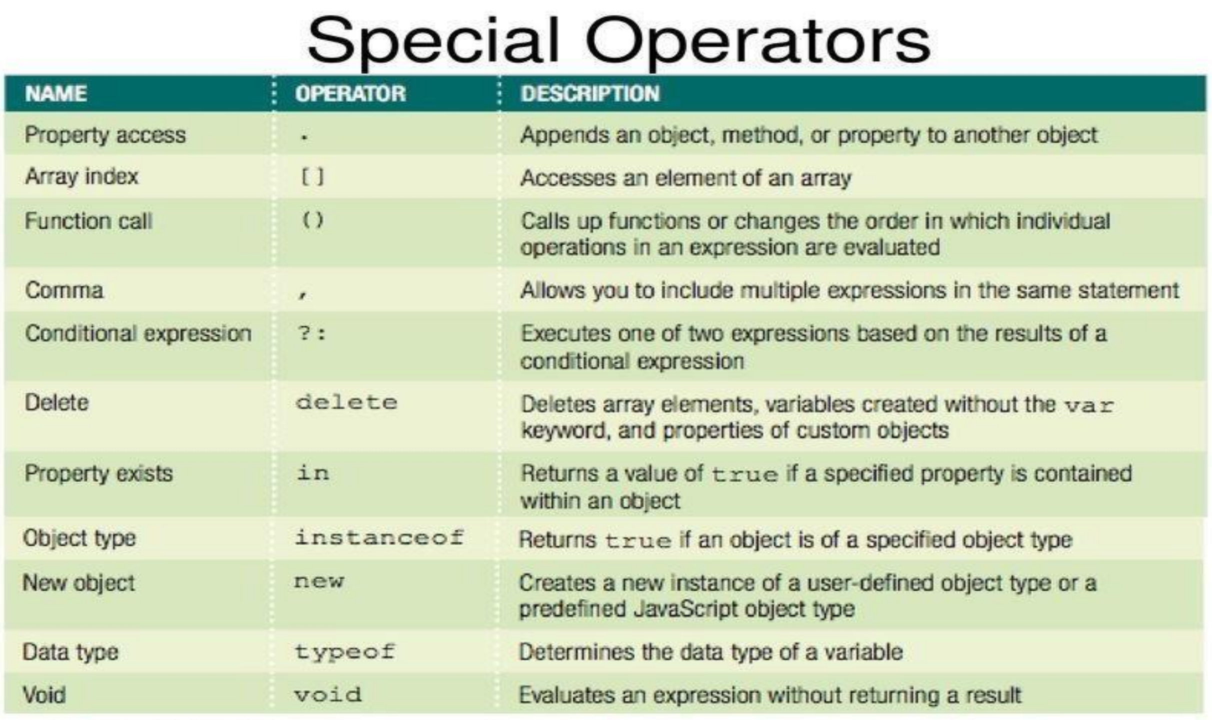
Logical Operators

Operator	Meaning	Example	Result
&&	Logical and	(5<2)&&(5>3)	False
||	Logical or	(5<2)||(5>3)	True
!	Logical not	!(5<2)	True

Assignment Operators



Operator	Example	Same As	Result
=	x=y		x=5
+=	x+=y	x=x+y	x=15
-=	x-=y	x=x-y	x=5
*=	x*=y	x=x*y	x=50
/=	x/=y	x=x/y	x=2
%=	x%=y	x=x%y	x=0



Special Operators

NAME	OPERATOR	DESCRIPTION
Property access	.	Appends an object, method, or property to another object
Array index	[ ]	Accesses an element of an array
Function call	( )	Calls up functions or changes the order in which individual operations in an expression are evaluated
Comma	,	Allows you to include multiple expressions in the same statement
Conditional expression	? :	Executes one of two expressions based on the results of a conditional expression
Delete	`delete`	Deletes array elements, variables created without the `var` keyword, and properties of custom objects
Property exists	`in`	Returns a value of `true` if a specified property is contained within an object
Object type	`instanceof`	Returns `true` if an object is of a specified object type
New object	`new`	Creates a new instance of a user-defined object type or a predefined JavaScript object type
Data type	`typeof`	Determines the data type of a variable
Void	`void`	Evaluates an expression without returning a result

Conditional Statements

There are three forms of if statement in JavaScript.

1. If Statement
2. If else statement
3. if else if statement

If statement

It evaluates the content only if expression is true. The signature of JavaScript if statement is given below.

```
        if(expression)
        {
        //content to be evaluated
        }
<script>
var a=20;
if(a>10){
document.write("value of a is greater than 10");
}
</script>
```

Output of the above example

```
value of a is greater than 10
```

If...else Statement

It evaluates the content whether condition is true of false. The syntax of JavaScript if-else statement is given below.

```
if(expression)
{
     //content to be evaluated if condition is true
}
else
{
        //content to be evaluated if condition is false
}
<script>
var a=20;
if(a%2==0)
{
        document.write("a is even number");
}
else
{
        document.write("a is odd number");
}
</script>
```

Output of the above example

```
a is even number
```

If...else if statement

It evaluates the content only if expression is true from several expressions.

The signature of JavaScript if else if statement is given below.

```
if(expression1)
{
//content to be evaluated if expression1 is true
}
else if(expression2)
{
//content to be evaluated if expression2 is true
}
else if(expression3)
{
//content to be evaluated if expression3 is true
}
else
{
//content to be evaluated if no expression is true
}
```

Let’s see the simple example of if else if statement in javascript.

```
<script>
var a=20;
if(a==10)
{
document.write("a is equal to 10");
}
else if(a==15){  document.write("a is equal to 15");
}
```

```
else if(a==20){
document.write("a is equal to 20");
}
else{
document.write("a is not equal to 10, 15 or 20");
}
</script>
```

Output :

a is equal to 20

Loops

loops are used to iterate the piece of code using for, while, do while or for-in loops. It makes the code compact. It is mostly used in array.

There are four types of loops in JavaScript.

1. for loop
2. while loop
3. do-while loop

For loop

The **for loop** iterates the elements for the fixed number of times. It should be used if number of iteration is known. The syntax of for loop is given below.

```
for (initialization; condition; increment)
{
   code to be executed
}
<script>
for (i=1; i<=5; i++)
{
document.write(i + "<br/>")
}
</script>
```

Output:

```
1
2
3
4
5
```

while loop

The **while loop** iterates the elements for the infinite number of times. It should be used if number of iteration is not known. The syntax of while loop is given below.

```
while (condition)
{
   code to be executed
}
```

Let’s see the simple example of while loop in javascript.

```
<script>
var i=11;
while (i<=15)
{
document.write(i + "<br/>");
i++;
}
</script>
```

Output:

```
11
12
13
14
15
```

do while loop

The **do while loop** iterates the elements for the infinite number of times like while loop. But, code is executed at least once whether condition is true or false. The syntax of do while loop is given below.

```
do{
 code to be executed
}while (condition);
<script>
var i=21;
do{
document.write(i + "<br/>");
i++;
}while (i<=25);
</script>
```

Output:

```
21
22
23
24
25
```

Arrays

array **is an object that represents a collection of similar type of elements.**
The syntax of creating array

```
var arrayname=new Array();
```

Here, **new keyword** is used to create instance of array.
Let's see the example of creating array directly.

```
<script>
var i;
var emp = new Array();
emp[0] = "Arun";
emp[1] = "Varun";
emp[2] = "John";
for (i=0;i<emp.length;i++){
document.write(emp[i] + "<br>");
}
</script>
```

Output of the above example

```
Arun
Varun
John
```

Functions

functions are used to perform operations. We can call JavaScript function many times to reuse the code.

Advantage of JavaScript function

There are mainly two advantages of JavaScript functions.

1. **Code reusability**: We can call a function several times so it save coding.
2. **Less coding**: It makes our program compact. We don’t need to write many lines of code each time to perform a common task.

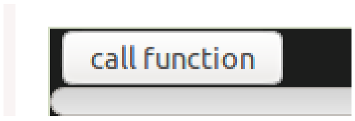
Function Syntax

The syntax of declaring function is given below.

```
1. function functionName([arg1, arg2, ...argN]){
2.  //code to be executed
3. }
```

Let’s see the simple example of function in JavaScript that does not has arguments.

```
    <script>
    function msg(){
    alert("hello! this is message");
    }
    </script>
<input type="button" onclick="msg()" value="call function"/>
```



Function Arguments

We can call function by passing arguments. Let’s see the example of function that has one argument.

```
<script>
function getcube(number){
alert(number*number*number);
}
</script>
<form>
<input type="button" value="click" onclick="getcube(4)"/>
                                    </form>
```

Function with Return Value

We can call function that returns a value and use it in our program. Let’s see the example of function that returns value.

```
<script>
function getInfo(){
return "hello javatpoint! How r u?";
}
</script>
<script>
document.write(getInfo());
</script>
```

o/p:hello javatpoint! How r u?

JavaScript Objects

A javaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc.

JavaScript is an object-based language. Everything is an object in JavaScript.

JavaScript is template based not class based. Here, we don't create class to get the object. But, we direct create objects.

Creating Objects

By creating instance of Object

The syntax of creating object directly is given below:

```
1. var objectname=new Object();
```

Here, **new keyword** is used to create object.

Let’s see the example of creating object directly.

```
1. <script>
2. var emp=new Object();
3. emp.id=101;
4. emp.name="Ravi Malik";
5. emp.salary=50000;
6. document.write(emp.id+" "+emp.name+" "+emp.salary);
7. </script>
   o/p:101 Ravi 50000
```

Accessing objects

A common way to access the property of an object is the dot property accessor syntax:

expression.identifier

expression should evaluate to an object, and identifier is the name of the property you'd like to access.

Predefined objects

Fundamental objects:

- Object
- Function
- Boolean
- Symbol

Error objects

Error objects are a special type of fundamental object. They include the basic Error type, as well as several specialized error types.

- Error
- AggregateError
- EvalError
- RangeError
- ReferenceError

Number and date objects

These are the base objects representing numbers, dates, and mathematical calculations.

- Number
- BigInt
- Math
- Date

Text processing objects

These objects represent strings and support manipulating them.

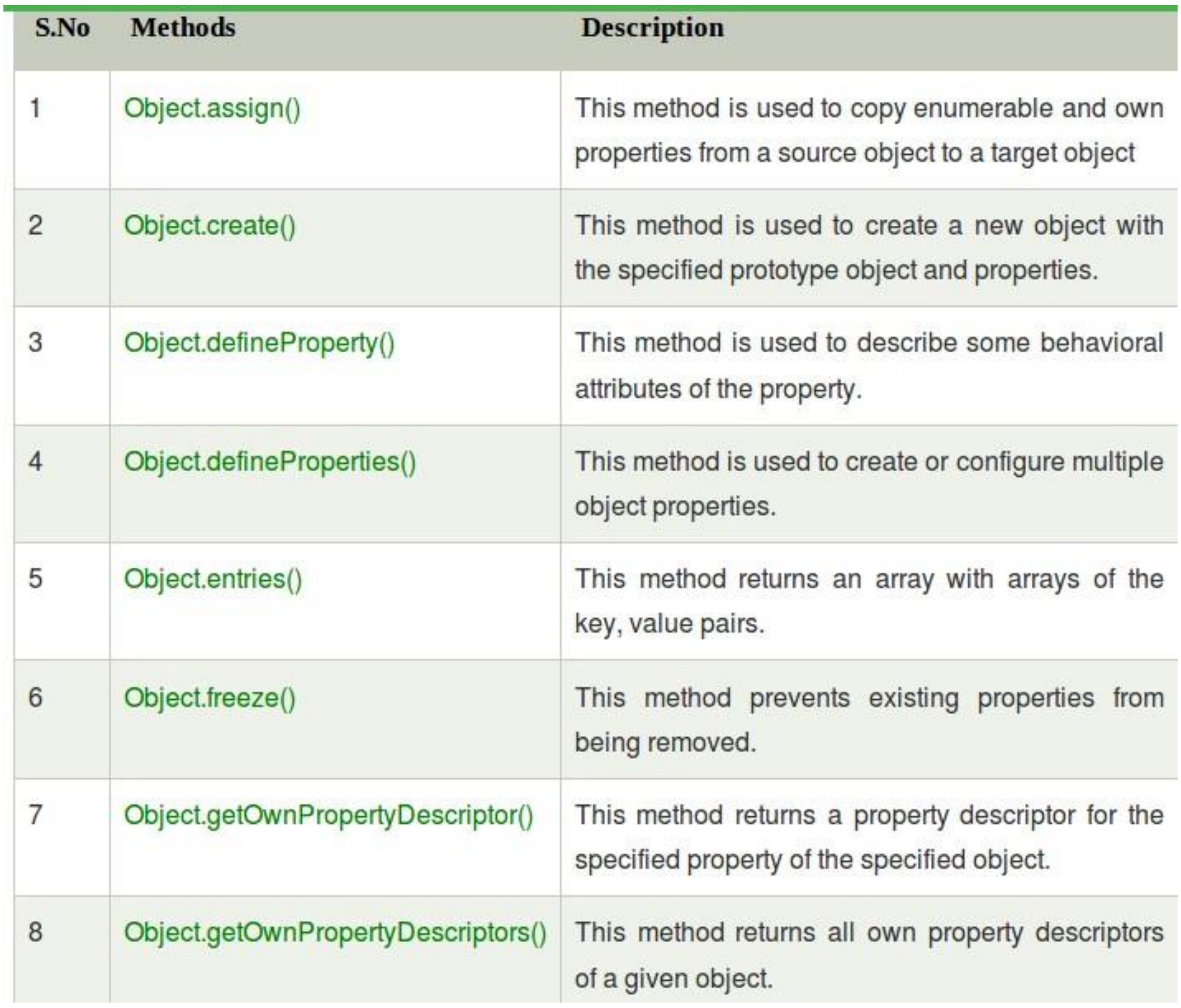
- String
- RegExp

Keyed collections

These objects represent collections which use keys. The iterable collections (Map and Set) contain elements which are easily iterated in the order of insertion.

- Map
- Set
- WeakMap
- WeakSet

Object methods



S.No	Methods	Description
1	Object.assign()	This method is used to copy enumerable and own properties from a source object to a target object
2	Object.create()	This method is used to create a new object with the specified prototype object and properties.
3	Object.defineProperty()	This method is used to describe some behavioral attributes of the property.
4	Object.defineProperties()	This method is used to create or configure multiple object properties.
5	Object.entries()	This method returns an array with arrays of the key, value pairs.
6	Object.freeze()	This method prevents existing properties from being removed.
7	Object.getOwnPropertyDescriptor()	This method returns a property descriptor for the specified property of the specified object.
8	Object.getOwnPropertyDescriptors()	This method returns all own property descriptors of a given object.

UNIT- II
JavaScript programming of reactive web pages elements: JavaScript Events- Mouse events, Keyboard events, Form events, window events, Event handlers, Frames, Form object, JavaScript Form Validation.

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:

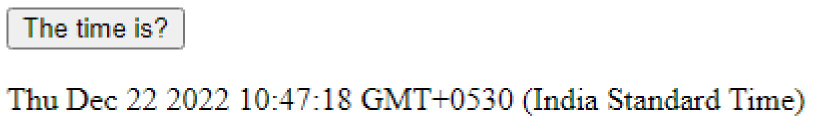
```
<element event='some JavaScript'>
```

With double quotes:

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

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

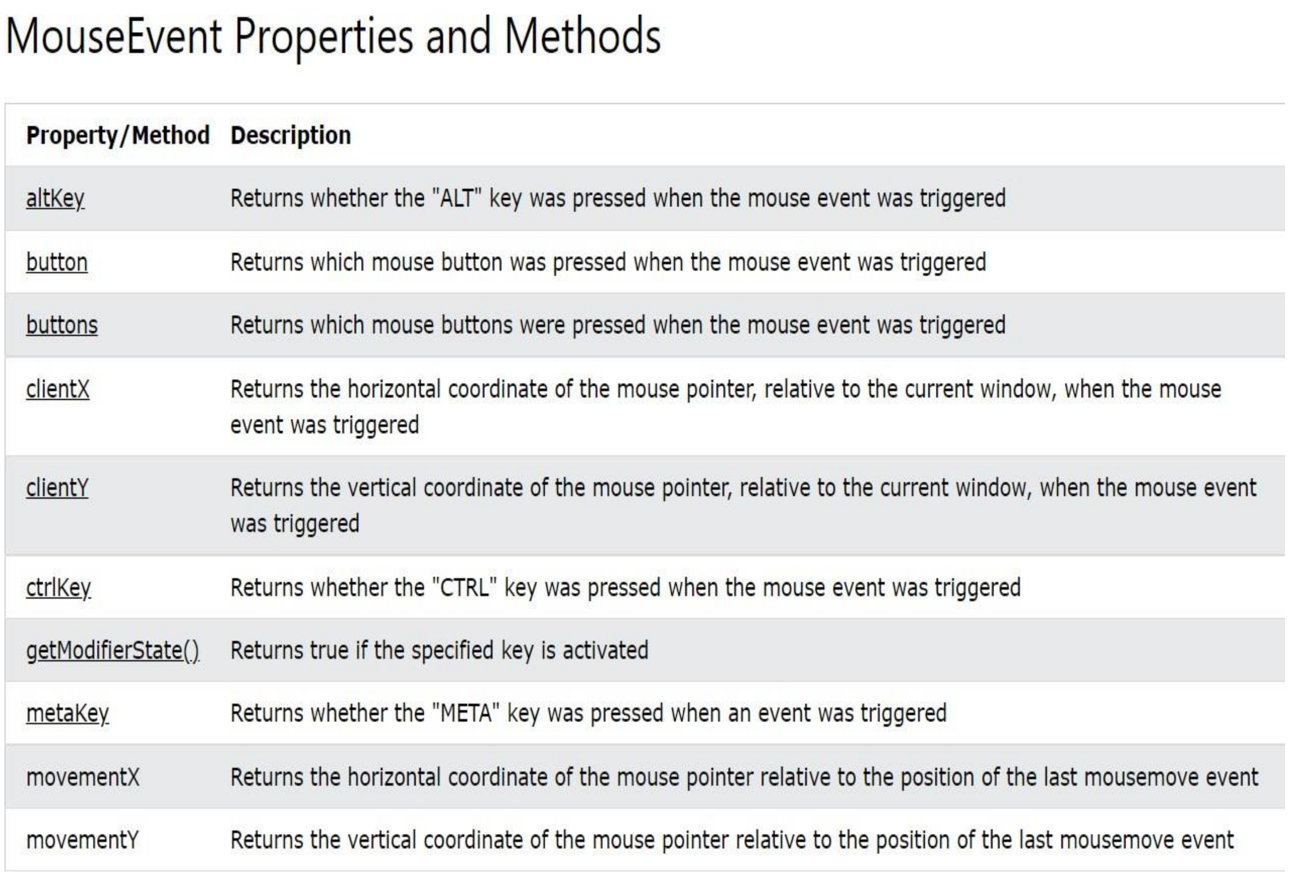
```
<button onclick="document.getElementById('demo').innerHTML = Date()">The time is?</button>
```



```
Thu Dec 22 2022 10:47:18 GMT+0530 (India Standard Time)
```

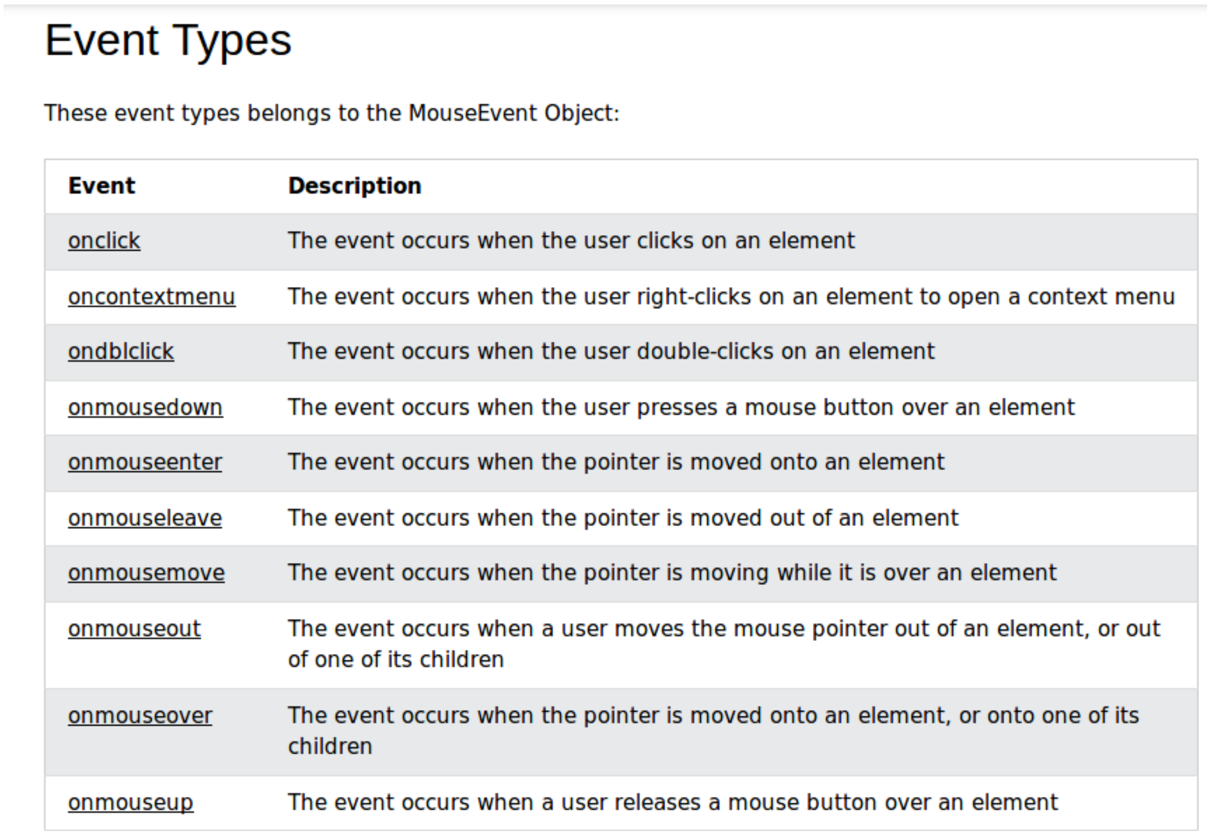
Mouse events:

Events that occur when the mouse interacts with the HTML document belongs to the MouseEvent Object.



MouseEvent Properties and Methods

Property/Method	Description
altKey	Returns whether the "ALT" key was pressed when the mouse event was triggered
button	Returns which mouse button was pressed when the mouse event was triggered
buttons	Returns which mouse buttons were pressed when the mouse event was triggered
clientX	Returns the horizontal coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered
clientY	Returns the vertical coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered
ctrlKey	Returns whether the "CTRL" key was pressed when the mouse event was triggered
getModifierState()	Returns true if the specified key is activated
metaKey	Returns whether the "META" key was pressed when an event was triggered
movementX	Returns the horizontal coordinate of the mouse pointer relative to the position of the last mousemove event
movementY	Returns the vertical coordinate of the mouse pointer relative to the position of the last mousemove event



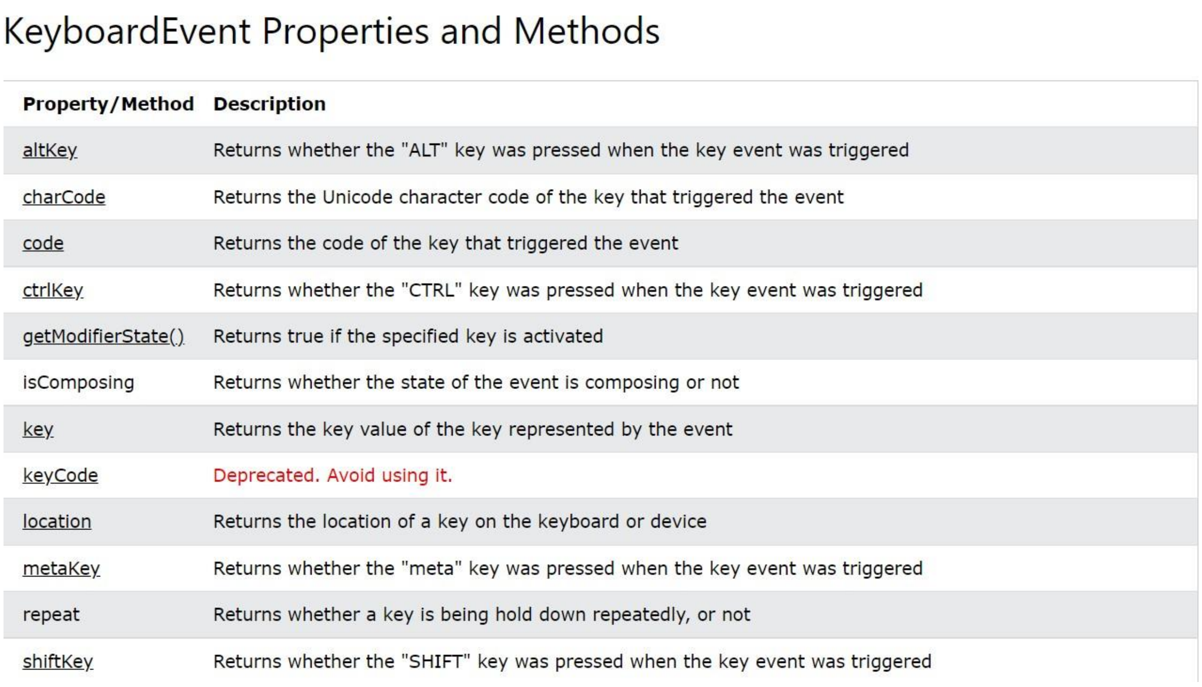
Event Types

These event types belongs to the MouseEvent Object:

Event	Description
onclick	The event occurs when the user clicks on an element
oncontextmenu	The event occurs when the user right-clicks on an element to open a context menu
ondblclick	The event occurs when the user double-clicks on an element
onmousedown	The event occurs when the user presses a mouse button over an element
onmouseenter	The event occurs when the pointer is moved onto an element
onmouseleave	The event occurs when the pointer is moved out of an element
onmousemove	The event occurs when the pointer is moving while it is over an element
onmouseout	The event occurs when a user moves the mouse pointer out of an element, or out of one of its children
onmouseover	The event occurs when the pointer is moved onto an element, or onto one of its children
onmouseup	The event occurs when a user releases a mouse button over an element

Keyboard events

Events that occur when user presses a key on the keyboard, belongs to the KeyboardEvent Object.



KeyboardEvent Properties and Methods

Property/Method	Description
altKey	Returns whether the "ALT" key was pressed when the key event was triggered
charCode	Returns the Unicode character code of the key that triggered the event
code	Returns the code of the key that triggered the event
ctrlKey	Returns whether the "CTRL" key was pressed when the key event was triggered
getModifierState()	Returns true if the specified key is activated
isComposing	Returns whether the state of the event is composing or not
key	Returns the key value of the key represented by the event
keyCode	Deprecated. Avoid using it.
location	Returns the location of a key on the keyboard or device
metaKey	Returns whether the "meta" key was pressed when the key event was triggered
repeat	Returns whether a key is being hold down repeatedly, or not
shiftKey	Returns whether the "SHIFT" key was pressed when the key event was triggered

Event Types

These event types belongs to the KeyboardEvent Object:

Event	Description
onkeydown	The event occurs when the user is pressing a key
onkeypress	The event occurs when the user presses a key
onkeyup	The event occurs when the user releases a key

Form events:

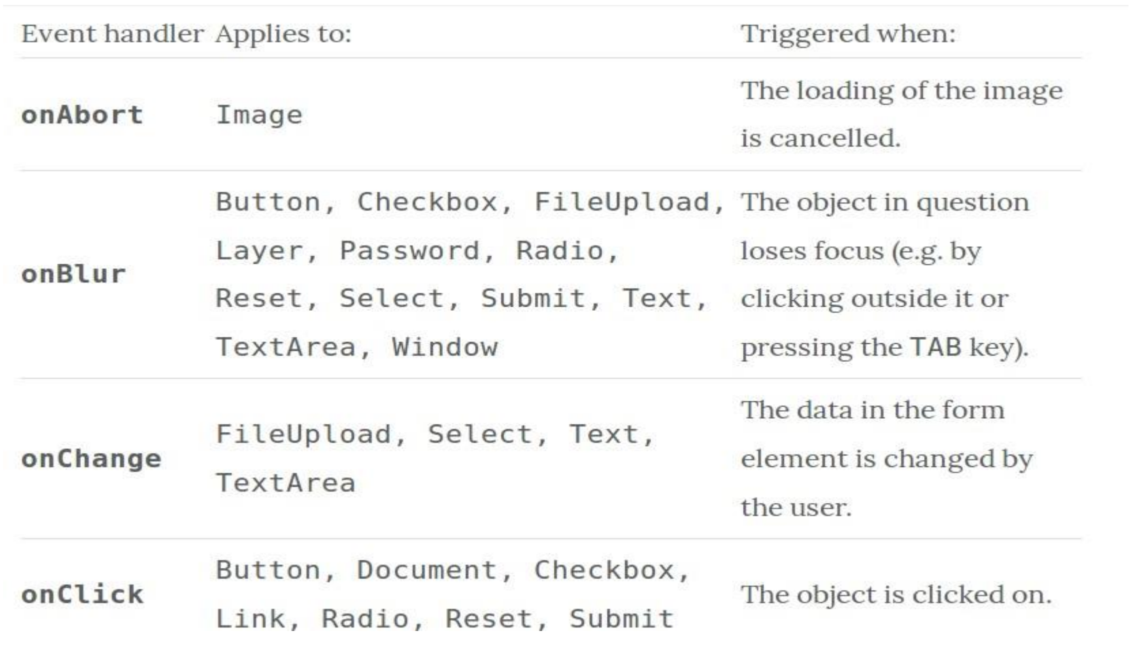
Event Performed	Event Handler	Description
focus	onfocus	When the user focuses on an element
submit	onsubmit	When the user submits the form
blur	onblur	When the focus is away from a form element
change	onchange	When the user modifies or changes the value of a form element

Window/Document events

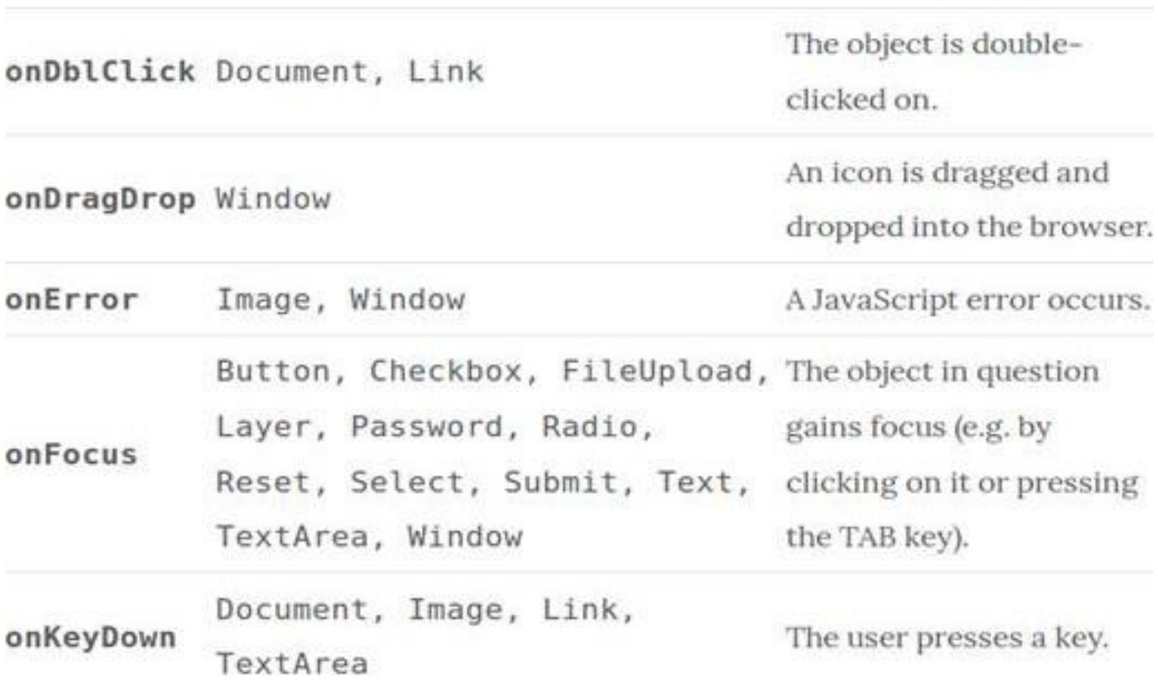
Event Performed	Event Handler	Description
load	onload	When the browser finishes the loading of the page
unload	onunload	When the visitor leaves the current webpage, the browser unloads it
resize	onresize	When the visitor resizes the window of the browser

Event handlers

To allow you to run your bits of code when these events occur, JavaScript provides us with event handlers. All the event handlers in JavaScript start with the word on, and each event handler deals with a certain type of event. Here's a list of all the event handlers in JavaScript, along with the objects they apply to and the events that trigger them:



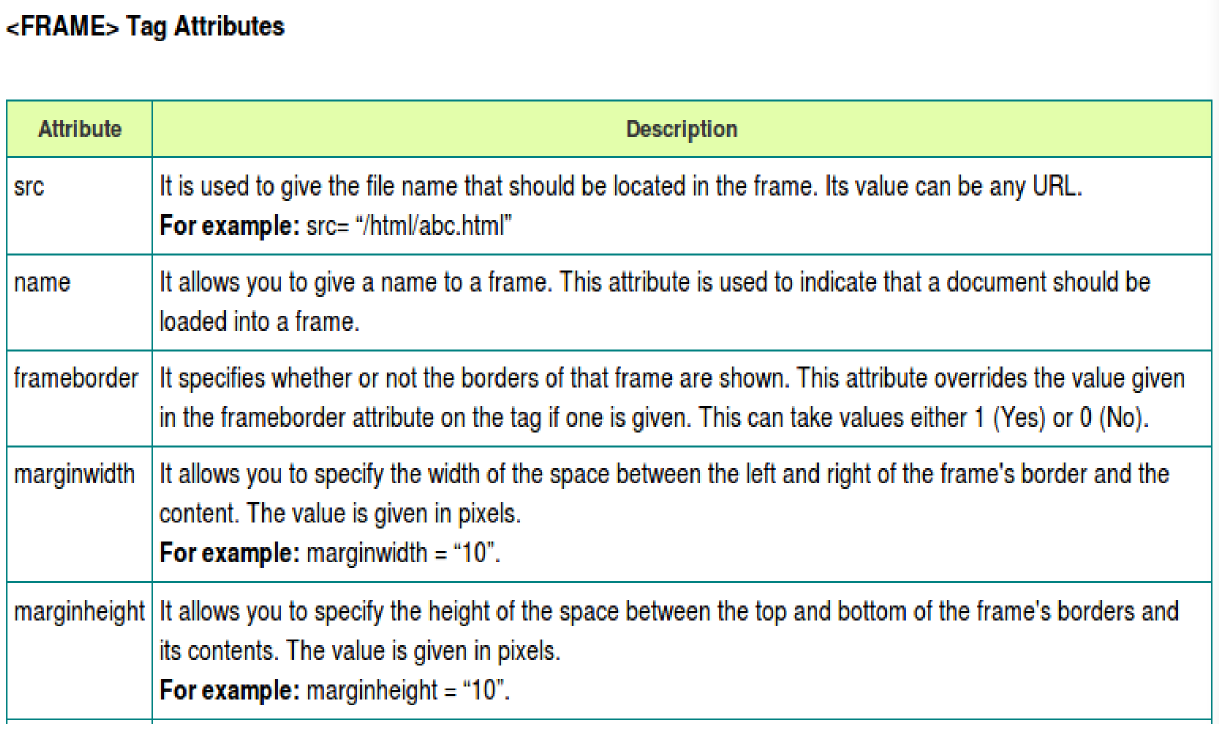
Event handler	Applies to:	Triggered when:
`onAbort`	`Image`	The loading of the image is cancelled.
`onBlur`	`Button, Checkbox, FileUpload, Layer, Password, Radio, Reset, Select, Submit, Text, TextArea, Window`	The object in question loses focus (e.g. by clicking outside it or pressing the TAB key).
`onChange`	`FileUpload, Select, Text, TextArea`	The data in the form element is changed by the user.
`onClick`	`Button, Document, Checkbox, Link, Radio, Reset, Submit`	The object is clicked on.



`onDblClick`	`Document, Link`	The object is double-clicked on.
`onDragDrop`	`Window`	An icon is dragged and dropped into the browser.
`onError`	`Image, Window`	A JavaScript error occurs.
`onFocus`	`Button, Checkbox, FileUpload, Layer, Password, Radio, Reset, Select, Submit, Text, TextArea, Window`	The object in question gains focus (e.g. by clicking on it or pressing the TAB key).
`onKeyDown`	`Document, Image, Link, TextArea`	The user presses a key.

Frames

- Frame object represents an HTML frame which defines one particular window(frame) within a frameset.
- It defines the set of frame that make up the browser window.
- Itisapropertyofthewindowobject.
  Syntax:<frame>
- It has no end tag but they need to be closed properly.
- It is an HTML element.
- It defines a particular area in which another HTML document can be displayed.
- A frame should be used within a <FRAMESET> tag.



<FRAME> Tag Attributes

Attribute	Description
src	It is used to give the file name that should be located in the frame. Its value can be any URL. **For example:** src= "/html/abc.html"
name	It allows you to give a name to a frame. This attribute is used to indicate that a document should be loaded into a frame.
frameborder	It specifies whether or not the borders of that frame are shown. This attribute overrides the value given in the frameborder attribute on the tag if one is given. This can take values either 1 (Yes) or 0 (No).
marginwidth	It allows you to specify the width of the space between the left and right of the frame's border and the content. The value is given in pixels. **For example:** marginwidth = "10".
marginheight	It allows you to specify the height of the space between the top and bottom of the frame's borders and its contents. The value is given in pixels. **For example:** marginheight = "10".

Form Object

form object is a Browser object of JavaScript used to access an HTML form. If a user wants to access all forms within a document then he can use the forms array. The form object is actually a property of document object that is uniquely created by the browser for each form present in a document. The properties and methods associated with form object are used to access the form fields, attributes and controls associated with forms.

Properties of Form Object:

- action
- elements[]
- encoding
- length
- method
- name
- target
- button
- checkbox
- FileUpload
- hidden
- password

action:

action property of form object is used to access the action attribute present in HTML associated with the <form> tag. This property is a read or write property and its value is a string.

elements[]:

elements property of form object is an array used to access any element of the form. It contains all fields and controls present in the form. The user can access any element associated with the form by using the looping concept on the elements array.

encoding:

The encoding property of a form object is used to access the enctype attribute present in HTML associated with the <form> tag. This property is a read or write property and its value is a string. This property helps determine the way of encoding the form data.

length:

length property of form object is used to specify the number of elements in the form. This denotes the length of the elements array associated with the form.

method:

method property of form object is used to access the method attribute present in HTML associated with the <form> tag. This property is a read or write property and its value is a string. This property helps determine the method by which the form is submitted.

name:

name property of form object denotes the form name.

target:

target property of form object is used to access the target attribute present in HTML associated with the <form> tag. This property denotes the name of the target window to which form it is to be submitted into.

button:

The button property of form object denotes the button GUI control placed in the form.

checkbox:

checkbox property of form object denotes the checkbox field placed in the form.

FileUpload:

FileUpload property of form object denotes the file upload field placed in the form..

hidden:

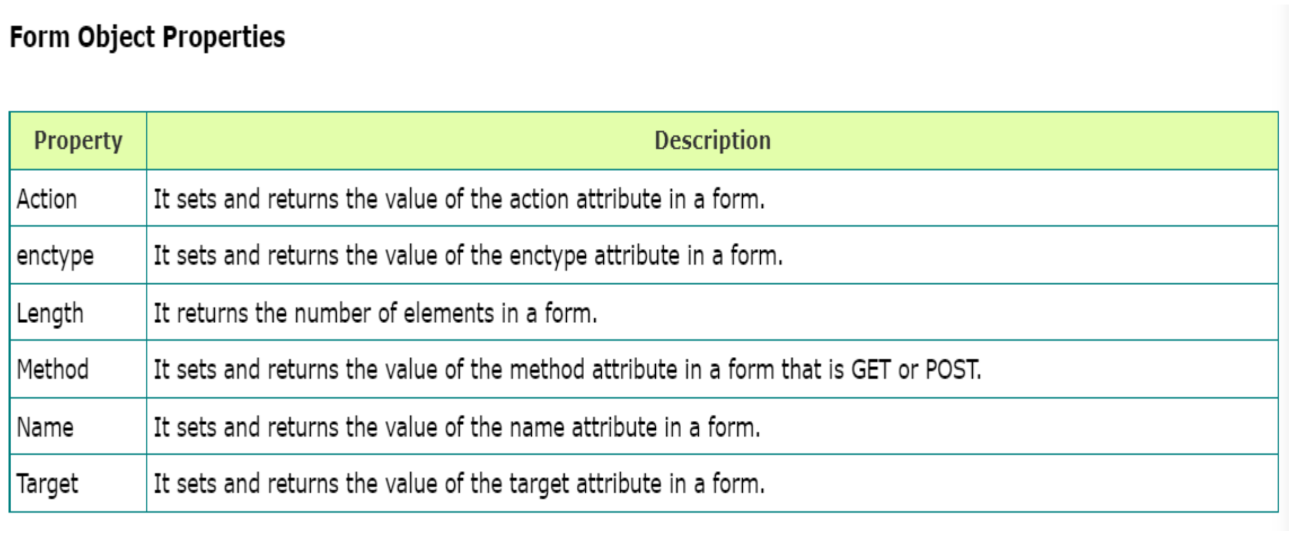
The hidden property of form object denotes the hidden field placed in the form.

password:

password property of form object denotes the object that is placed as a password field in the form.

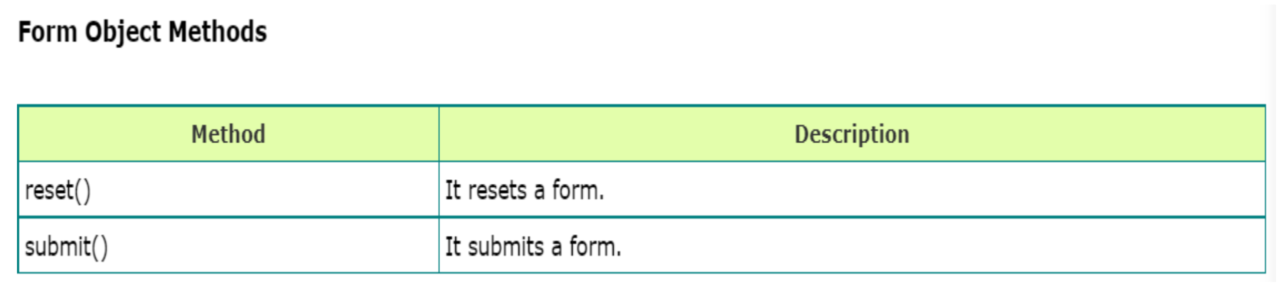
Form Object

- Form object represents an HTML form.
- It is used to collect user input through elements like text fields, check box and radio button, select option, text area, submit buttons and etc.



Form Object Properties

Property	Description
Action	It sets and returns the value of the action attribute in a form.
enctype	It sets and returns the value of the enctype attribute in a form.
Length	It returns the number of elements in a form.
Method	It sets and returns the value of the method attribute in a form that is GET or POST.
Name	It sets and returns the value of the name attribute in a form.
Target	It sets and returns the value of the target attribute in a form.

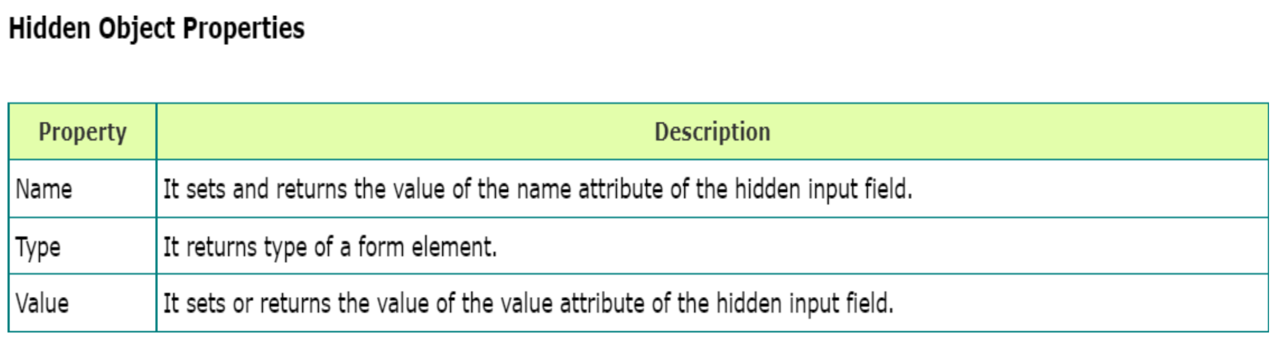


Form Object Methods

Method	Description
reset()	It resets a form.
submit()	It submits a form.

Hidden Object

- Hidden object represents a hidden input field in an HTML form and it is invisible to the user.
- This object can be placed anywhere on the web page.
- It is used to send hidden form of data to a server.



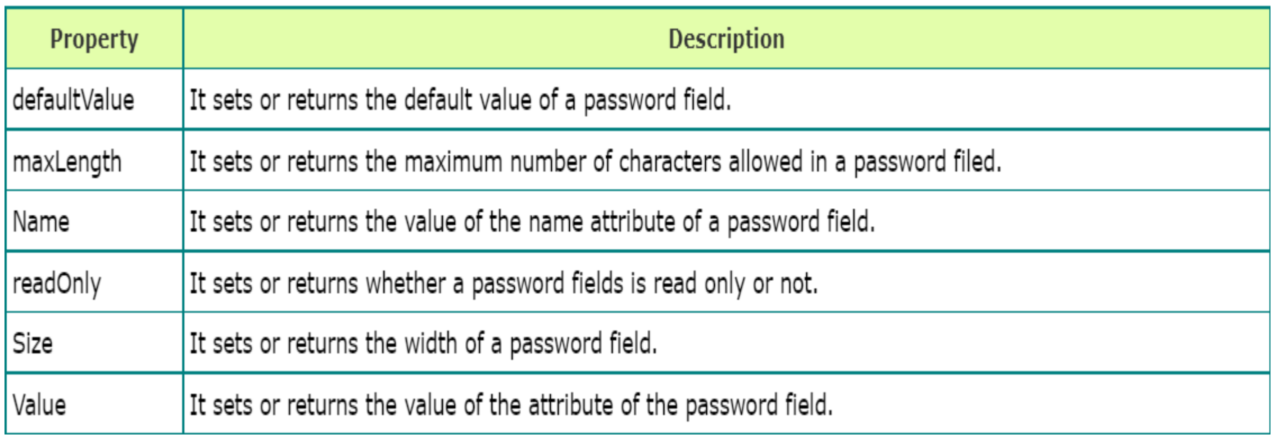
Hidden Object Properties

Property	Description
Name	It sets and returns the value of the name attribute of the hidden input field.
Type	It returns type of a form element.
Value	It sets or returns the value of the value attribute of the hidden input field.

Password Object

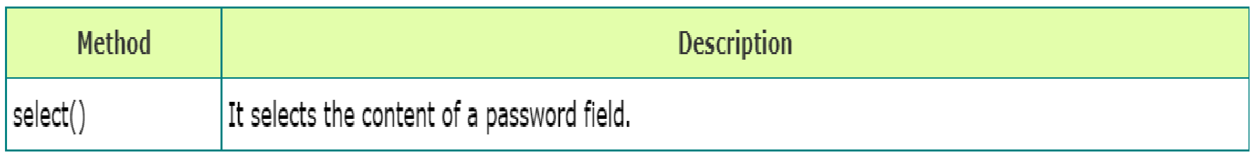
- Password object represents a single-line password field in an HTML form.
- The content of a password field will be masked – appears as spots or asterisks in the browser using password object.

Password Object Properties



Property	Description
defaultValue	It sets or returns the default value of a password field.
maxLength	It sets or returns the maximum number of characters allowed in a password filed.
Name	It sets or returns the value of the name attribute of a password field.
readOnly	It sets or returns whether a password fields is read only or not.
Size	It sets or returns the width of a password field.
Value	It sets or returns the value of the attribute of the password field.

Password Object Methods

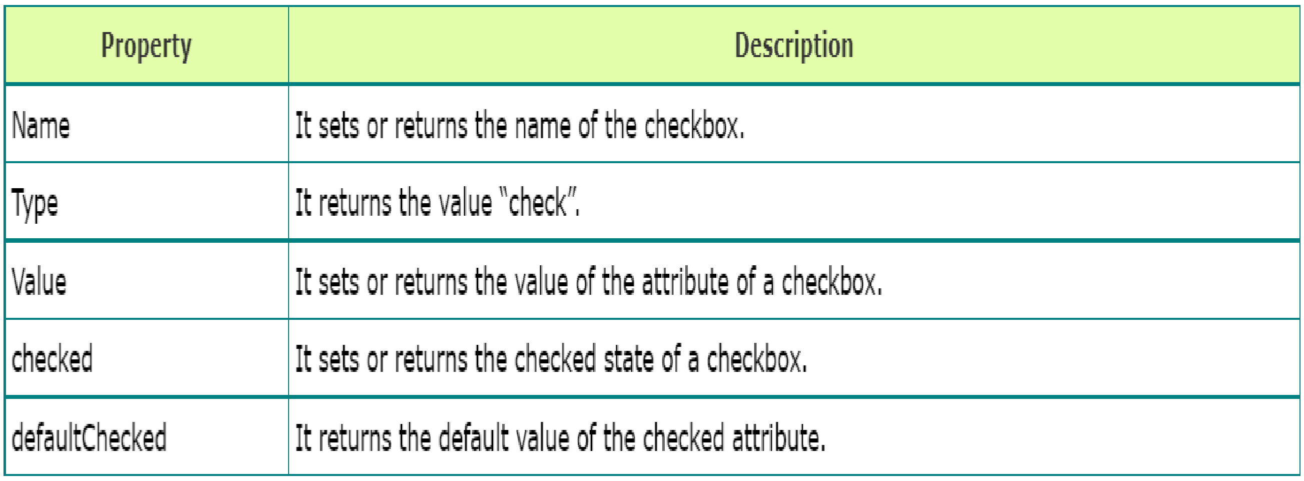


Method	Description
select()	It selects the content of a password field.

Checkbox Object

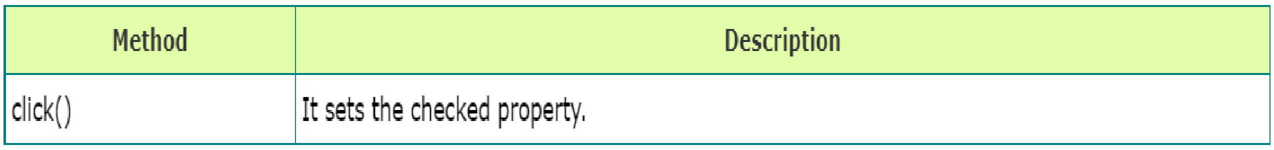
- Check box object represents a checkbox in an HTML form.
- It allows the user to select one or more options from the available choices.

Checkbox Object Properties



Property	Description
Name	It sets or returns the name of the checkbox.
Type	It returns the value "check".
Value	It sets or returns the value of the attribute of a checkbox.
checked	It sets or returns the checked state of a checkbox.
defaultChecked	It returns the default value of the checked attribute.

Checkbox Object Methods

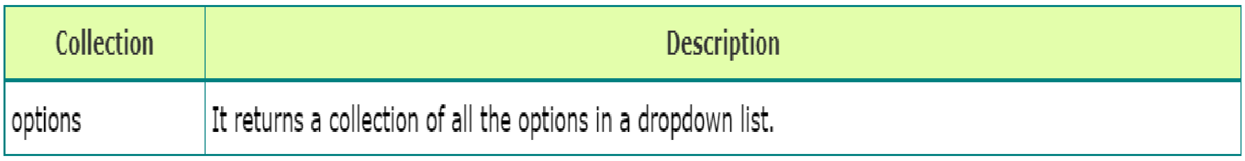


Method	Description
click()	It sets the checked property.

Select Object

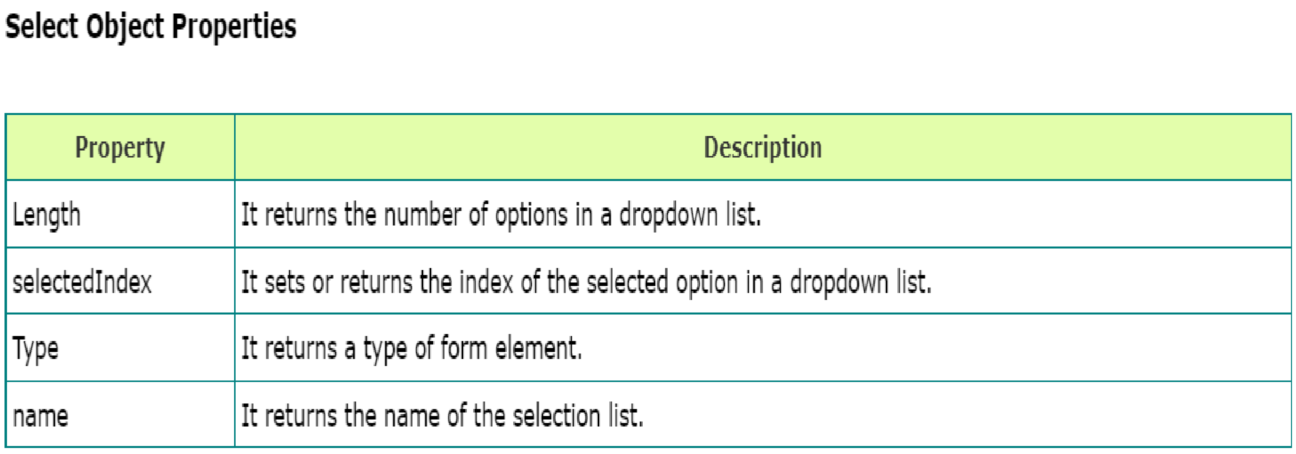
- Select object represents a dropdown list in an HTML form.
- It allows the user to select one or more options from the available choices.

Select Object Collections



Collection	Description
options	It returns a collection of all the options in a dropdown list.

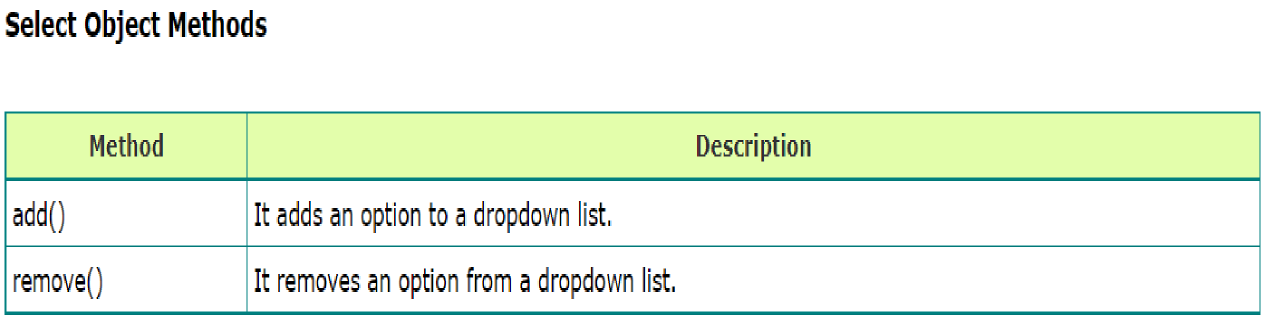
Select Object Properties



Select Object Properties

Property	Description
Length	It returns the number of options in a dropdown list.
selectedIndex	It sets or returns the index of the selected option in a dropdown list.
Type	It returns a type of form element.
name	It returns the name of the selection list.

Select Object Methods



Select Object Methods

Method	Description
add()	It adds an option to a dropdown list.
remove()	It removes an option from a dropdown list.

Option Object

- Option object represents an HTML <option> element.
- It is used to add items to a select element.

Option Object Properties

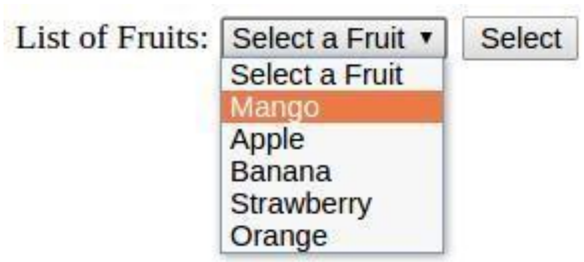
Property	Description
Index	It sets or returns the index position of an option in a dropdown list.
Text	It sets or returns the text of an option element.
defaultSelected	It determines whether the option is selected by default.
Value	It sets or returns the value to the server if the option was selected.
Prototype	It is used to create additional properties.

Option Object Methods

Methods	Description
blur()	It removes the focus from the option.
focus()	It gives the focus to the option.

```
<html>
<head>
    <script type="text/javascript">
    function optionfruit(select)
    {
        var a = select.selectedIndex;
        var fav = select.options[a].value;
        if(a==0)
        {
            alert("Please select a fruit");
        }
        else
        {
            document.write("Your Favorite Fruit is <b>"+fav+".</b>");
        }
    }
    </script>
</head>
<body>
    <form>
        List of Fruits:
        <select name="fruit">
            <option value="0">Select a Fruit</option>
            <option value="Mango">Mango</option>
            <option value="Apple">Apple</option>
            <option value="Banana">Banana</option>
            <option value="Strawberry">Strawberry</option>
            <option value="Orange">Orange</option>
        </select>
        <input type="button" value="Select" onClick="optionfruit(this.form.fruit);">
    </form>
</body>
</html>
```

Output:



Your Favorite Fruit is **Mango.**

Radio Object

Radio object represents a radio button in an HTML form.

Radio Object Properties

Property	Description
Checked	It sets or returns the checked state of a radio button.
defaultChecked	Returns the default value of the checked attribute.
Name	It sets or returns the value of the name attribute of a radio button.
Type	It returns the type of element which is radio button.
Value	It sets or returns the value of the radio button.

Radio Object Methods

Method	Description
blur()	It takes the focus away from the radio button.
click()	It acts as if the user clicked the button.
focus()	It gives the focus to the radio button.

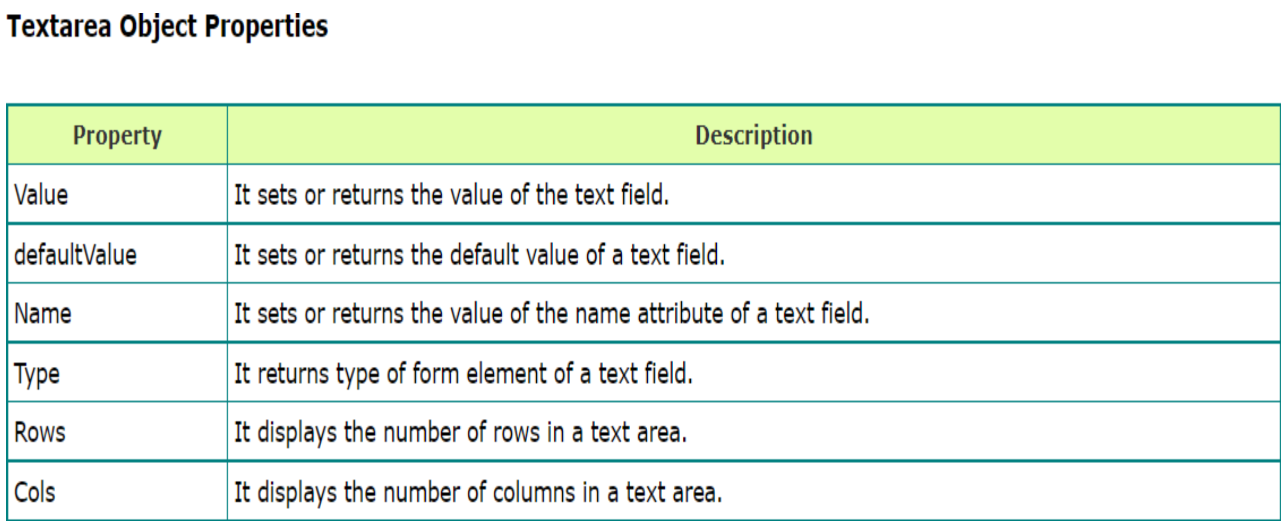
Text Object

Text object represents a single-line text input field in an HTML form.

Text Object Properties

Property	Description
Value	It sets or returns the value of the text field.
defaultValue	It sets or returns the default value of a text field.
Name	It sets or returns the value of the name attribute of a text field.
maxLength	It sets or returns the maximum number of characters allowed in a text field.
readOnly	It sets or returns whether a text field is read-only or not.
Size	It sets or returns the width of a text field.
Type	It returns type of form element of a text field.

Textarea Object
Textarea object represents a text-area in an HTML form.



Textarea Object Properties

Property	Description
Value	It sets or returns the value of the text field.
defaultValue	It sets or returns the default value of a text field.
Name	It sets or returns the value of the name attribute of a text field.
Type	It returns type of form element of a text field.
Rows	It displays the number of rows in a text area.
Cols	It displays the number of columns in a text area.

JavaScript Form Validation
HTML form validation can be done by JavaScript.

If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

```
functionvalidateForm(){
 letx=document.forms["myForm"]["fname"].value;
 if(x==""){
   alert("Namemustbefilledout");
   returnfalse;
  }
}
```

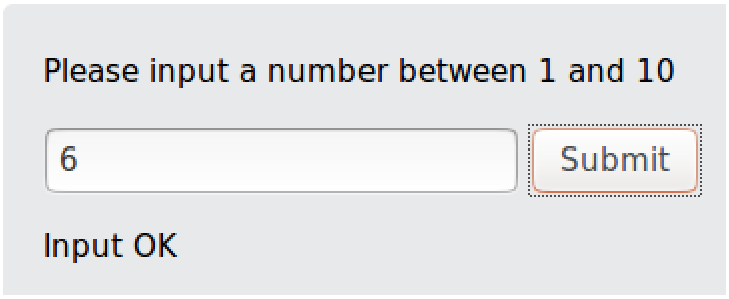
The function can be called when the form is submitted:

HTML Form Example

<formname="myForm"action="/action_page.php"**onsubmit="return validateForm()"**method="post">
Name:<inputtype="text"name="fname">
<inputtype="submit"value="Submit">
</form>

JavaScript Can Validate Numeric Input

JavaScript is often used to validate numeric input:


Please input a number between 1 and 10

6

Input OK

UNIT- III
PERL: Data Types, Variables, Scalars, Operators, Conditional statements ,Loops, Arrays , Strings , Hashes , Lists , Built-in Functions, Pattern matching and regular expression operators.

Perl is a general purpose, high level interpreted and dynamic programming language. Perl supports both the procedural and Object-Oriented programming. Perl is a lot similar to C syntactically and is easy for the users who have knowledge of C, C++. Since Perl is a lot similar to other widely used languages syntactically, it is easier to code and learn in Perl. Programs can be written in Perl in any of the widely used text editors like Notepad++, gedit, etc.

Data types
There are different data types available in Perl for different purposes. Some of them have been presented below:

- Boolean
- Integer
- Float
- Array
- String

Unlike other languages such as C++ or Java, Perl does not require defining variables along with a specific data type. The type of a variable is picked based on the value assigned to it.
Here is a brief overview of the types:

Boolean
Boolean data type is used to store **true** or **false** values. The numeric value 0 is used to represent **false**, whereas any other numeric value represents **true**. Let's look at the code below:
$false = 0; # reutrns false
$true = 1; # any values greater or less than 0 returns true

Integer
An integer is a positive or negative whole number. Perl allows you to assign integer constants in decimal, hexadecimal, octal or binary numbering systems. Consider the following code:
$negative = -3; # negative
$zero = 0; # zero (can also be false, if used as a Boolean
$positive = 123; # positive decimal
$zeroPos = 0123; #0 prefix is used to sepcify octal - octal value = 83 decimal $hex = 0xAB; #0x prefix is used to specify hexadecimal - hexadecimal value = 171 decimal
$bin = 0b1010; # 0b prefix is used to specify binary - binary value = 10 decimal print $negative," " ,$zero," " , $positive," " , $zeroPos," " , $hex," " , $bin;

Output

```
-3 0 123 83 171 10
```

Float

Floating point numbers, doubles or simply called floats are decimal numbers.

```
$float1 = 1.23;
$float2 = 10.0000001;
print $float1," ",$float2;
```

Output

```
1.23 10.0000001
```

Array

An array is like a list of values (similar or of different data types). The simplest form of an array is indexed by an integer, and ordered by the index, with the first element lying at **index 0**. We use @ to initiate an array with values enclosed in () pair of parenthesis. Look at the following code:

```
@intarray = (1, 2, 3);
# An array of integers print "@intarray \n";
@floatarray = (1.123, 2.356, 19.76);
# An array of floats print "@floatarray \n";
@chararray = ('a', 'b','c');
# An array of characters print "@chararray \n";
@mixed = (1, 2, 3, 'a', 'b', 'c');
#contains both characters and numbers print "@mixed";
```

Output

```
1 2 3 1.123 2.356 19.76 a b c 1 2 3 a b c
```

String

A string is an array of characters. We can declare a string using either single quotes (') or double quotes (").

```
$string1 = "A quick brown fox jumps over the lazy dog";
print $string1;
```

Output

```
A quick brown fox jumps over the lazy dog
```

Variables

A variable in any programming language is a named piece of computer memory to hold some program data. Variables are an essential part of a computer program.

You can declare a variable in Perl using a $ sign followed by its name, e.g., $myVar.

onsider the following Perl code where we store data in variables and display them on the screen.

```
$string = "This is a string.";
# stores string
```

```
$int = 5;
# stores an integer
$float = 5.7;
# stores a floating point type value
 $char = 'a';
# stores character type value
print $string, "\n";
print "An integer type: ", $int, "\n";
print "A float type: ", $float, "\n";
print "A character type: ", $char, "\n";
```

Output

```
This is a string. An integer type: 5 A float type: 5.7 A character type: a
```

scalars

A scalar is a single unit of data. That data might be an integer number, floating point, a character, a string, a paragraph, or an entire web page.

Here is a simple example of using scalar variables −

```
#!/usr/bin/perl
$age = 25;             # An integer assignment
$name = "John Paul";   # A string
$salary = 1445.50;     # A floating point
print "Age = $age\n";
print "Name = $name\n";
print "Salary = $salary\n";
```

This will produce the following result −

```
Age = 25
Name = John Paul
Salary = 1445.5
```

Numeric Scalars

A scalar is most often either a number or a string. Following example demonstrates the usage of various types of numeric scalars −

```
#!/usr/bin/perl
$integer = 200;
$negative = -300;
$floating = 200.340;
$bigfloat = -1.2E-23;
# 377 octal, same as 255 decimal
$octal = 0377;
# FF hex, also 255 decimal
```

```
$hexa = 0xff;
print "integer = $integer\n";
print "negative = $negative\n";
print "floating = $floating\n";
print "bigfloat = $bigfloat\n";
print "octal = $octal\n";
print "hexa = $hexa\n";
```

This will produce the following result −

```
integer = 200
negative = -300
floating = 200.34
bigfloat = -1.2e-23
octal = 255
hexa = 255
```

String Scalars

Following example demonstrates the usage of various types of string scalars. Notice the difference between single quoted strings and double quoted strings −

```
#!/usr/bin/perl
$var = "This is string scalar!";
$quote = 'I m inside single quote - $var';
$double = "This is inside single quote - $var";
$escape = "This example of escape -\tHello, World!";
print "var = $var\n";
print "quote = $quote\n";
print "double = $double\n";
print "escape = $escape\n";
```

This will produce the following result −

```
var = This is string scalar!
quote = I m inside single quote - $var
double = This is inside single quote - This is string scalar!
escape = This example of escape -       Hello, World
```

Scalar Operations

You will see a detail of various operators available in Perl in a separate chapter, but here we are going to list down few numeric and string operations.

```
#!/usr/bin/perl
$str = "hello" . "world";       # Concatenates strings.
$num = 5 + 10;                  # adds two numbers.
$mul = 4 * 5;                   # multiplies two numbers.
$mix = $str . $num;             # concatenates string and number.
print "str = $str\n";
print "num = $num\n";
print "mul = $mul\n";
print "mix = $mix\n";
```

This will produce the following result –

```
str = helloworld
num = 15
mul = 20
mix = helloworld15
```

Multiline Strings

If you want to introduce multiline strings into your programs, you can use the standard single quotes as below –

```
#!/usr/bin/perl
$string = 'This is
a multiline
string';
print "$string\n";
```

This will produce the following result –

```
This is
a multiline
string
```

Operators

4 + 5 is equal to 9. Here 4 and 5 are called operands and + is called operator. Perl language supports many operator types, but following is a list of important and most frequently used operators –

- Numeric operators
- String operators
- Logical operators
- Bitwise operators
- Special operators
- Comparison operators
- Assignment operators

Numeric operators

Numeric operators are the standard arithmetic operators like addition (+), subtraction (-), multiplication (*), division (/) and modulo (%), etc.

String operators

String operators are positive and negative regular expression with repetition (=~ and !~) and concatenation (.).

String Concatenation operator

```
use 5.010;
use strict;
use warnings;
my $result = "Hello this is " . "JavaTpoint.";
say $result;
```

output:

```
Hello this is JavaTpoint.
```

String Repetition operator

```
use 5.010;
use strict;
use warnings;
my $result = "Thank You " x 3;
say $result;
```

output:

```
Thank You Thank You Thank You.
```

Here, note that on the right of 'x' it must be an integer.

There should be space on either side of the 'x' operator.

For example,

$result = "Thank You " x 3; # This is correct

1. $result = "Thank You "x3; # This is incorrect

Logical operators

Logical operators give a Boolean value to their operands. They are (&&, || and or).

&& -> In && operator, if $a is 0, then value of $a && $b must be false irrespective of the value of $b. So perl does not bother to check $b value. This is called short-circuit evaluation.

|| -> In || operator, if $a is non-zero, then value of $a && $b must be true irrespective of the value of $b. So perl does not bother to check $b value.

Example:

```
use 5.010;
use strict;
use warnings;
$a = 0;
$b = 12;
my $result1 = $a && $b;
say $result1;
$a = 12;
$b = 14;
```

```
        my $result2 = $a || $b;
        say $result2;
```

Output:

```
0
12
```

Bitwise operators

Bitwise operators treat their operands numerically at bit level. These are (<<, >>, &, |, ^, <<=, >>=, &=, |=, ^=).

Every number will be denoted in terms of 0s and 1s. Initially integers will be converted into binary bit and result will be evaluated. Final result will be displayed in the integer form.

Example:

```
        use 5.010;
        use strict;
        use warnings;
        #OR operator
        my $result1 = 124.3 | 99;
        say $result1;
        #AND operator
        my $result2 = 124.3 & 99;
        say $result2;
        #XOR operator
        my $result3 = 124.3 ^ 99;
        say $result3;
        #Shift operator
        my $result4 = 124 >> 3;
        say $result4;
```

Output:

```
127
96
31
15
```

Special operators

The auto-increment (++) operator is a special operator that increments the numeric character itself by 1.

Example:

```
        use 5.010;
        use strict;
        use warnings;
```

```
    my $num = 9;
    my $str = 'x';
    $num++;
    $str++;
    say $num++;
    say $str++;
```

Output:

```
10
Y
```

Comparison operators

The comparison operator compares the values of its operands. These are (==, <, <=, >, >=, <=>, !=).

Example:

```
    use 5.010;
    use strict;
    use warnings;
    say "Enter your salary:";
    my $salary = <>;
    if($salary >= 20000)
    {
        say "You are earning well";
    } else {
      say "You are not earning well";
    }
```

Output:

```
Enter your salary:
15000
You are not earning well
```

Assignment operators

The assignment operator assigns a value to a variable.

These are (=, +=, -=, *=, /=, |=, &=, %=)

Example:

```
use 5.010;
    use strict;
    use warnings;
    $a = 20;
    my $result1 = $a += $a;
    say $result1;
    my $result2 = $a -= 10;
    say $result2;
    my $result3 = $a |= 10;
    say $result3;
   my $result4 = $a &= 10;
   say $result4;
output:
40
30
30
10
```

Conditional Statements

Perl conditional statements helps in the decision making, which require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

A Perl if statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an **if** statement in Perl programming language is −

```
if(boolean_expression) {
   # statement(s) will execute if the given condition is true
}
```

If the boolean expression evaluates to **true** then the block of code inside the **if** statement will be executed. If boolean expression evaluates to **false** then the first set of code after the end of the **if** statement (after the closing curly brace) will be executed.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

```
#!/usr/local/bin/perl
$a = 10;
# check the boolean condition using if statement
if( $a < 20 ) {
   # if condition is true then print the following
   printf "a is less than 20\n";
}
print "value of a is : $a\n";
$a = "";
```

```
# check the boolean condition using if statement
if( $a ) {
   # if condition is true then print the following
   printf "a has a true value\n";
}
print "value of a is : $a\n";
```

o/p:

```
a is less than 20
value of a is : 10
value of a is :
```

if-else

A Perl **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax

The syntax of an **if...else** statement in Perl programming language is −

```
if(boolean_expression) {
   # statement(s) will execute if the given condition is true
} else {
   # statement(s) will execute if the given condition is false
}
```

If the boolean expression evaluates to **true**, then the **if block** of code will be executed otherwise **else block** of code will be executed.

The number 0, the strings '0' and "" , the empty list () , and undef are all **false** in a boolean context and all other values are **true**. Negation of a true value by **!** or **not** returns a special false value.

```
#!/usr/local/bin/perl
$a = 100;
# check the boolean condition using if statement
if( $a < 20 ) {
   # if condition is true then print the following
   printf "a is less than 20\n";
} else {
   # if condition is false then print the following
   printf "a is greater than 20\n";
}
print "value of a is : $a\n";
$a = "";
# check the boolean condition using if statement
if( $a ) {
   # if condition is true then print the following
   printf "a has a true value\n";
} else {
   # if condition is false then print the following
   printf "a has a false value\n";
}
print "value of a is : $a\n";
```

When the above code is executed, it produces the following result −

```
a is greater than 20
value of a is : 100
a has a false value
value of a is :
```

if-elsif-else

An **if** statement can be followed by an optional **elsif...else** statement, which is very useful to test the various conditions using single if...elsif statement.

When using **if , elsif , else** statements there are few points to keep in mind.

- An **if** can have zero or one **else**'s and it must come after any **elsif**'s.
- An **if** can have zero to many **elsif**'s and they must come before the **else**.
- Once an **elsif** succeeds, none of the remaining **elsif**'s or **else**'s will be tested.

Syntax

The syntax of an **if...elsif...else** statement in Perl programming language is −

```
if(boolean_expression 1) {
   # Executes when the boolean expression 1 is true
} elsif( boolean_expression 2) {
   # Executes when the boolean expression 2 is true
} elsif( boolean_expression 3) {
   # Executes when the boolean expression 3 is true
} else {
   # Executes when the none of the above condition is true
}
#!/usr/local/bin/perl
$a = 100;
# check the boolean condition using if statement
if( $a == 20 ) {
   # if condition is true then print the following
   printf "a has a value which is 20\n";
} elsif( $a ==  30 ) {
   # if condition is true then print the following
   printf "a has a value which is 30\n";
} else {
   # if none of the above conditions is true
   printf "a has a value which is $a\n";
}
        o/p:a has a value which is 100
```

Loops

Perl programming language provides the following types of loop to handle the looping requirements.

A while loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Perl programming language is −

```
while(condition) {
   statement(s);
}
```

```
#!/usr/local/bin/perl
$a = 10;
# while loop execution
while( $a < 20 ) {
   printf "Value of a: $a\n";
   $a = $a + 1;
}
```

Here we are using the comparison operator < to compare value of variable $a against 20. So while value of $a is less than 20, **while** loop continues executing a block of code next to it and as soon as the value of $a becomes equal to 20, it comes out. When executed, above code produces the following result −

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
Value of a: 19
```

An until loop statement in Perl programming language repeatedly executes a target statement as long as a given condition is false.

Syntax

The syntax of an **until** loop in Perl programming language is −

```
until(condition) {
   statement(s);
}
```

```
#!/usr/local/bin/perl
$a = 5;
# until loop execution
until( $a > 10 ) {
   printf "Value of a: $a\n";
   $a = $a + 1;
}
```

Here we are using the comparison operator > to compare value of variable $a against 10. So until the value of $a is less than 10, **until** loop continues executing a block of code next to it and as soon as the value of $a becomes greater than 10, it comes out. When executed, above code produces the following result −

```
Value of a: 5
Value of a: 6
Value of a: 7
Value of a: 8
Value of a: 9
Value of a: 10
```

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in Perl programming language is −

```
for ( init; condition; increment ) {
   statement(s);
}
```

```
#!/usr/local/bin/perl
# for loop execution
for( $a = 10; $a < 20; $a = $a + 1 ) {
   print "value of a: $a\n";
}
```

When the above code is executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

The syntax of a **do...while** loop in Perl is −

```
do {
   statement(s);
}while( condition );
```

```
#!/usr/local/bin/perl
$a = 10;
# do...while loop execution
do{
   printf "Value of a: $a\n";
   $a = $a + 1;
}while( $a < 20 );
```

When the above code is executed, it produces the following result −

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
value of a: 19
```

Arrays

An array is a variable that stores an ordered list of scalar values. Array variables are preceded by an "at" (@) sign. To refer to a single element of an array, you will use the dollar sign ($) with the variable name followed by the index of the element in square brackets.

Here is a simple example of using the array variables −

```
#!/usr/bin/perl
@ages = (25, 30, 40);
@names = ("John Paul", "Lisa", "Kumar");
print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

Here we have used the escape sign (\) before the $ sign just to print it. Other Perl will understand it as a variable and will print its value. When executed, this will produce the following result −

```
$ages[0] = 25
$ages[1] = 30
$ages[2] = 40
$names[0] = John Paul
$names[1] = Lisa
$names[2] = Kumar
```

In Perl, List and Array terms are often used as if they're interchangeable. But the list is the data, and the array is the variable.

Array Creation

Array variables are prefixed with the @ sign and are populated using either parentheses or the qw operator. For example −

```
@array = (1, 2, 'Hello');
@array = qw/This is an array/;
```

The second line uses the qw// operator, which returns a list of strings, separating the delimited string by white space. In this example, this leads to a four-element array; the first element is 'this' and last (fourth) is 'array'. This means that you can use different lines as follows −

```
@days = qw/Monday
Tuesday
...
Sunday/;
```

Accessing Array Elements

When accessing individual elements from an array, you must prefix the variable with a dollar sign ($) and then append the element index within the square brackets after the name of the variable. For example −

```
#!/usr/bin/perl
@days = qw/Mon Tue Wed Thu Fri Sat Sun/;
print "$days[0]\n";
print  "$days[1]\n";
print  "$days[2]\n";
print  "$days[6]\n";
print "$days[-1]\n";
print "$days[-7]\n";
```

This will produce the following result −

```
Mon
Tue
Wed
Sun
Sun
Mon
```

Array indices start from zero, so to access the first element you need to give 0 as indices. You can also give a negative index, in which case you select the element from the end, rather than the beginning, of the array. This means the following −

```
print $days[-1]; # outputs Sun
print $days[-7]; # outputs Mon
```

Sequential Number Arrays

Perl offers a shortcut for sequential numbers and letters. Rather than typing out each element when counting to 100 for example, we can do something like as follows −

```
#!/usr/bin/perl
@var_10 = (1..10);
@var_20 = (10..20);
@var_abc = (a..z);
print "@var_10\n";   # Prints number from 1 to 10
print "@var_20\n";   # Prints number from 10 to 20
print "@var_abc\n"; # Prints number from a to z
```

Here double dot (..) is called **range operator**. This will produce the following result −

```
1 2 3 4 5 6 7 8 9 10
10 11 12 13 14 15 16 17 18 19 20
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Hashes

A hash is a set of **key/value** pairs. Hash variables are preceded by a percent (%) sign. To refer to a single element of a hash, you will use the hash variable name preceded by a "$" sign and followed by the "key" associated with the value in curly brackets..

Here is a simple example of using the hash variables −

```
#!/usr/bin/perl
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
print "\$data{'John Paul'} = $data{'John Paul'}\n";
print "\$data{'Lisa'} = $data{'Lisa'}\n";
print "\$data{'Kumar'} = $data{'Kumar'}\n";
```

This will produce the following result −

```
$data{'John Paul'} = 45
$data{'Lisa'} = 30
$data{'Kumar'} = 40
```

Creating Hashes

Hashes are created in one of the two following ways. In the first method, you assign a value to a named key on a one-by-one basis −

```
$data{'John Paul'} = 45;
$data{'Lisa'} = 30;
$data{'Kumar'} = 40;
```

In the second case, you use a list, which is converted by taking individual pairs from the list: the first element of the pair is used as the key, and the second, as the value. For example −

```
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
```

For clarity, you can use => as an alias for , to indicate the key/value pairs as follows −

```
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
```

Here is one more variant of the above form, have a look at it, here all the keys have been preceded by hyphen (-) and no quotation is required around them −

```
%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);
```

But it is important to note that there is a single word, i.e., without spaces keys have been used in this form of hash formation and if you build-up your hash this way then keys will be accessed using hyphen only as shown below.

```
$val = %data{-JohnPaul}
$val = %data{-Lisa}
```

Accessing Hash Elements

When accessing individual elements from a hash, you must prefix the variable with a dollar sign ($) and then append the element key within curly brackets after the name of the variable. For example −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
```

```
print "$data{'John Paul'}\n";
print "$data{'Lisa'}\n";
print "$data{'Kumar'}\n";
```

This will produce the following result −

```
45
30
40
```

Extracting Slices

You can extract slices of a hash just as you can extract slices from an array. You will need to use @ prefix for the variable to store the returned value because they will be a list of values −

```
#!/uer/bin/perl
%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);
@array = @data{-JohnPaul, -Lisa};
print "Array : @array\n";
```

This will produce the following result −

```
Array : 45 30
```

Extracting Keys and Values

You can get a list of all of the keys from a hash by using **keys** function, which has the following syntax −

```
keys %HASH
```

This function returns an array of all the keys of the named hash. Following is the example −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@names = keys %data;
print "$names[0]\n";
print "$names[1]\n";
print "$names[2]\n";
```

This will produce the following result −

```
Lisa
John Paul
Kumar
```

Getting Hash Size

You can get the size - that is, the number of elements from a hash by using the scalar context on either keys or values. Simply saying first you have to get an array of either the keys or values and then you can get the size of array as follows −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@keys = keys %data;
```

```
$size = @keys;
print "1 - Hash size: is $size\n";
@values = values %data;
$size = @values;
print "2 - Hash size:  is $size\n";
```

This will produce the following result −

```
1 - Hash size: is 3
2 - Hash size: is 3
```

Add and Remove Elements in Hashes

Adding a new key/value pair can be done with one line of code using simple assignment operator. But to remove an element from the hash you need to use **delete** function as shown below in the example −

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@keys = keys %data;
$size = @keys;
print "1 - Hash size: is $size\n";
# adding an element to the hash;
$data{'Ali'} = 55;
@keys = keys %data;
$size = @keys;
print "2 - Hash size:  is $size\n";
# delete the same element from the hash;
delete $data{'Ali'};
@keys = keys %data;
$size = @keys;
print "3 - Hash size:  is $size\n";
```

This will produce the following result −

```
1 - Hash size: is 3
2 - Hash size: is 4
3 - Hash size: is 3
```

Strings

Strings are an essential part of the Perl language. They are scalar variables, so they start with ($) sign. A string can be defined within a single quote (') or double quote (").

Perl String Operators

The operators make it easy to manipulate a string in different ways. There are two types of string operators:

- Concatenation (.)
- Repetition (x)

Concatenation Operator

Perl strings are concatenated with a (.) sign instead of (+) sign.

```
$firstName = "Christian";
$lastName = "Grey";
$fullName = $firstName . " " . $lastName;
print "$fullName\n";
```

```
Christian Grey
```

Repeitition Operator

Perl strings can be repeated a number of times with (x) variable.

```
$text = "Thank You ";
$output = $text x 3;
print "$output\n";
```

Output:

```
Thank You Thank You Thank You
```

Initializing and Declaring a String

In Perl, to declare a string use **my** keyword before variable name.

A string can be initialised and declared with the following syntax:

```
my $variableName = "";
```

In this example, we have shown how to initialize and declare a string. We have printed several strings together by using a dot (.) operator.

```
se strict;
use warnings;
# Declaring and initializing a string.
my $msg1 = "Welcome at JavaTpoint.";
my $msg2 = "This is our Perl Tutorial.";
#printing using . operator.
print $msg1 . "" . $msg2. "\n";
#print as separate arguments.
print $msg1, "",$msg2, "\n";
```

```
#embedd string in a bigger string.
print "$msg1$msg2\n";
```

Output:

```
Welcome at JavaTpoint. This is our Perl Tutorial.
Welcome at JavaTpoint. This is our Perl Tutorial.
Welcome at JavaTpoint. This is our Perl Tutorial.
```

Determining String Length, length()

string length can be determined with length() function.

```
my $msg = "Our site javaTpoint provides all type of tutorials";
print "String Length : ", length($msg), "\n";
```

Output:

```
String Length : 50
```

Replacing a string with another string, s///g

A string can be replaced with another string in two ways.

In first one, we have replaced **Tigers** with **Lions** which occurs single time in the string with **s///.**

In second one, we have replaced **roses** with **flowers** globally with **s///g.**

```
my $var1 = "Tigers are big and frightening.";
$var1 =~ s/Tigers/Lions/;
print "$var1\n";
my $var2 = "Red roses are very popular. Yellow roses are less seen.";
$var2 =~ s/roses/flowers/g;
print "$var2\n";
```

Output:

```
Lions are big and frightening.
Red flowers are very popular. Yellow flowers are less seen.
```

Perl Concatenating two Strings (.=)

Two strings can be joined together using (.=) operator.

```
my $str1 = "Where there is a will,";
my $str2 = "there is a way.\n";
my $joining = '';
$joining = $str1 . ' ';
$joining .= $str2;
print $joining;
```

Output:

```
Where there is a will, there is a way.
```

Lists

A list is a collection of scalar values. We can access the elements of a list using indexes. Index starts with 0 (0th index refers to the first element of the list). We use parenthesis and comma operators to construct a list. In Perl, scalar variables start with a $ symbol whereas list variables start with @ symbol.

Example :

```
#!/usr/bin/perl
# Empty List assigned to an array
@empty_list = ();
# List of integers
@integer_list = (1, 2, 3);
# List of strings assigned to an array
@string_list = ("hai", "for", "hai");
print "Empty list: @empty_list\n";
print "Integer list: @integer_list\n";
print "String list: @string_list\n";
```

Output:

```
Empty list:
Integer list: 1 2 3
String list: hai for hai
```

Builtin functions

The chr() function in Perl returns a string representing a character whose Unicode code point is an integer.

join()function

join() function is used to combine the elements of a List into a single string with the use of a separator provided to separate each element.

```
#!/usr/bin/perl
# Initializing list with alphabets A to Z
@list = (A..Z);
# Printing the original list
print "List: @list\n";
# Using join function introducing
# hyphen between each alphabets
print "\nString after join operation:\n";
print join("-", @list);
```

Output:

```
List: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```
String after join operation:
A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P-Q-R-S-T-U-V-W-X-Y-Z
```

reverse()function

Reverse() function in Perl returns the elements of List in reverse order in a list context.

```
# Initializing a list
@list = ("Raj", "E123", 12000);
# Reversing the list
@rname = reverse(@list);
# Printing the reversed list
print "Reversed list is @rname";
# Initializing a scalar
$string = "for";
# Reversing a scalar
$r = reverse($string);
print "\nReversed string is $r";
```

```
Reversed list is 12000 E123 Raj
Reversed string is rof
```

Regular Expressions

A regular expression is a string of characters that defines the pattern or patterns you are viewing.

There are three regular expression operators within Perl.

- Match Regular Expression - m//
- Substitute Regular Expression - s///
- Transliterate Regular Expression - tr///

1. The Match Operator

The match operator, m//, is used to match a string or statement to a regular expression. For example, to match the character sequence "foo" against the scalar $bar, you might use a statement like this −

```
#!/usr/bin/perl
$bar = "This is foo and again foo";
if ($bar =~ /foo/) {
   print "First time is matching\n";
} else {
   print "First time is not matching\n";
}
$bar = "foo";
if ($bar =~ /foo/) {
   print "Second time is matching\n";
} else {
   print "Second time is not matching\n";
}
```

When above program is executed, it produces the following result −

```
First time is matching
Second time is matching
```

2. The Substitution Operator

The substitution operator, s///, is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is −

```
s/PATTERN/REPLACEMENT/;
#/user/bin/perl
$string = "The cat sat on the mat";
$string =~ s/cat/dog/;
print "$string\n";
```

When above program is executed, it produces the following result −

```
The dog sat on the mat
```

3. The Translation Operator

Translation is similar, but not identical, to the principles of substitution, but unlike substitution, translation (or transliteration) does not use regular expressions for its search on replacement values. The translation operators are −

```
tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds
#/user/bin/perl
$string = 'The cat sat on the mat';
$string =~ tr/a/o/;
print "$string\n";
```

When above program is executed, it produces the following result −

```
The cot sot on the mot.
```

UNIT -IV
PHP : Data Types, Variables, Operators, Conditional statements ,Loops ,Arrays - Indexed Array, Associative Array, String Functions, Functions- Parameterized Function, Call By Value, Call By Reference , File Handling, PHP Form handling.

The **PHP Hypertext Preprocessor (PHP)** is a programming language that allows web developers to create dynamic content that interacts with databases. PHP is basically used for developing web based software applications.

PHP is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning PHP:

- 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.

Characteristics of PHP

Five important characteristics make PHP's practical nature possible −

- Simplicity
- Efficiency
- Security
- Flexibility
- Familiarity

Applications of PHP

As mentioned before, PHP is one of the most widely used language over the web. I'm going to list few of them here:

- PHP performs system functions, i.e. from files on a system it can create, open, read, write, and close them.
- PHP can handle forms, i.e. gather data from files, save data to a file, through email you can send data, return data to the user.
- You add, delete, modify elements within your database through PHP.
- Access cookies variables and set cookies.
- Using PHP, you can restrict users to access some pages of your website.

- It can encrypt data.

Variables

Variables are "containers" for storing information.

Creating (Declaring) PHP Variables

In PHP, a variable starts with the $ sign, followed by the name of the variable:

Example

```
<?php
$txt="Helloworld!";
$x=5;
$y=10.5;
?>
```

output:

Hey

```
Helloworld!
5
10.5
```

Data Types

Variables can store data of different types, and different data types can do different things.

PHP supports the following data types:

- String
- Integer
- Float (floating point numbers - also called double)
- Boolean

String

A string is a sequence of characters, like "Hello world!".

A string can be any text inside quotes. You can use single or double quotes:

```
<?php
$x="Helloworld!";
$y='Helloworld!';
echo$x;
echo"<br>";
echo$y;
?>
```

```
Helloworld!
Hello world!
```

Integer

An integer data type is a non-decimal number between -2,147,483,648 and 2,147,483,647.

Rules for integers:

- An integer must have at least one digit
- An integer must not have a decimal point
- An integer can be either positive or negative
- Integers can be specified in: decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2) notation

In the following example $x is an integer. The PHP var_dump() function returns the data type and value:

```
<?php
$x=5985;
var_dump($x);
?>
```

```
int(5985)
```

Float

A float (floating point number) is a number with a decimal point or a number in exponential form.

In the following example $x is a float. The PHP var_dump() function returns the data type and value:

Example

```
<?php
$x=10.365;
var_dump($x);
?>
```

```
float(10.365)
```

Boolean

A Boolean represents two possible states: TRUE or FALSE.

```
$x=true;
$y = false;
```

Operators

4 + 5 is equal to 9. Here 4 and 5 are called operands and + is called operator.

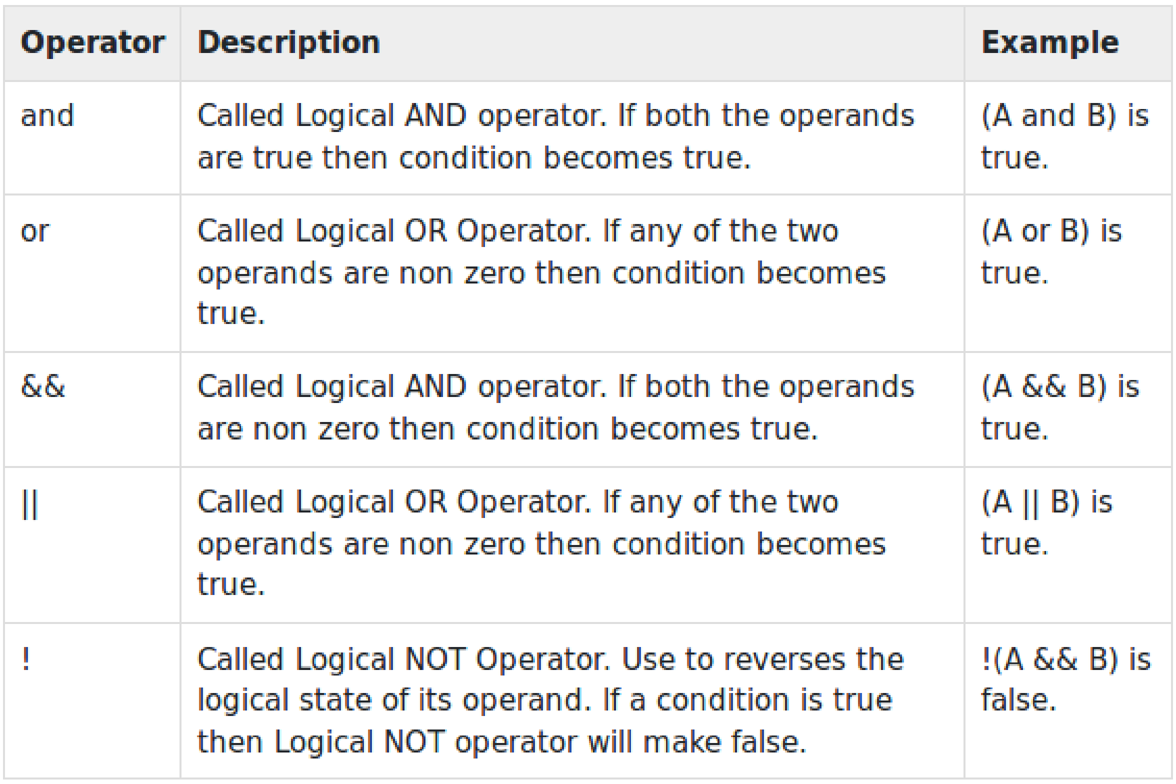
Arithmetic Operators

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Comparison Operators

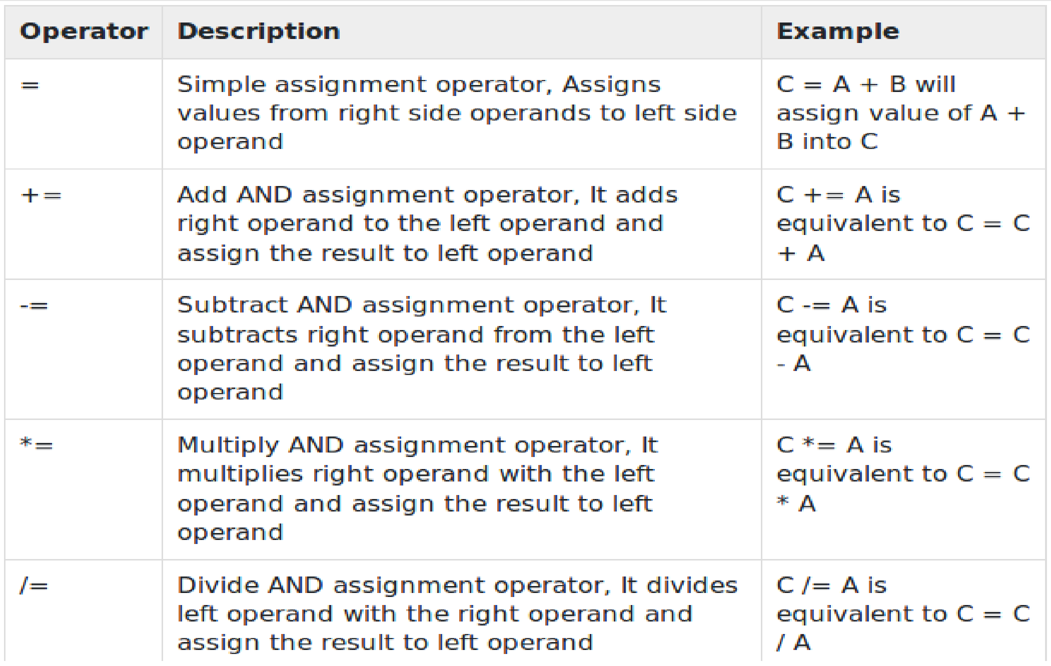
Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left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

Logical Operators



Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then condition becomes true.	(A and B) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(A or B) is true.
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(A || B) is true.
!	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 false.

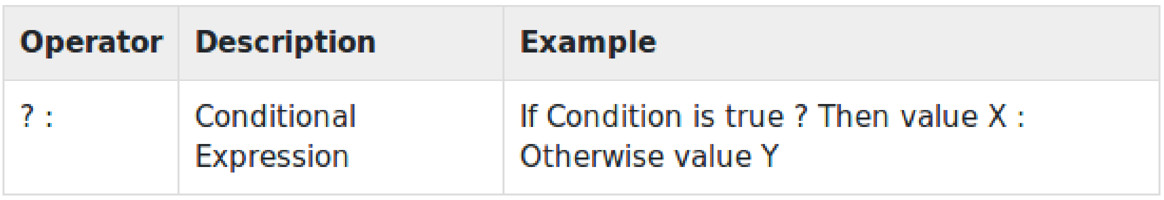
Assignment Operators



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

Conditional Operator

There is one more operator called conditional operator. This first evaluates an expression for a true or false value and then execute one of the two given statements



Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

depending upon the result of the evaluation.

Conditional statements

The If...Else Statement

If you want to execute some code if a condition is true and another code if a condition is false, use the if. ..else statement.

Syntax

```
if (condition)
  code to be executed if condition is true;
else
  code to be executed if condition is false;
```

The following example will output "Have a nice weekend!" if the current day is Friday, Otherwise, it will output "Have a nice day!":

```
<html>
  <body>
    <?php
      $d = date("D");
      if ($d == "Fri")
        echo "Have a nice weekend!";
      else
        echo "Have a nice day!";
    ?>
  </body>
</html>
```

It will produce the following result −

```
Have a nice weekend!
```

ElseIf Statement

If you want to execute some code if one of the several conditions are true use the elseif statement

Syntax

```
if (condition)
   code to be executed if condition is true;
elseif (condition)
   code to be executed if condition is true;
else
   code to be executed if condition is false;
```

Example

The following example will output "Have a nice weekend!" if the current day is Friday, and "Have a nice Sunday!" if the current day is Sunday. Otherwise, it will output "Have a nice day!"

```
<html>
   <body>
      <?php
         $d = date("D");
         if ($d == "Fri")
            echo "Have a nice weekend!";
         elseif ($d == "Sun")
            echo "Have a nice Sunday!";
         else
            echo "Have a nice day!";
      ?>
   </body>
</html>
```

It will produce the following result −

```
o/p:Have a nice Weekend!
```

Loops

Loops in PHP are used to execute the same block of code a specified number of times. PHP supports following four loop types.

- for − loops through a block of code a specified number of times.
- while − loops through a block of code if and as long as a specified condition is true.
- do...while − loops through a block of code once, and then repeats the loop as long as a special condition is true.
- foreach − loops through a block of code for each element in an array.

for loop

The for statement is used when you know how many times you want to execute a statement or a block of statements.

Syntax

```
for (initialization; condition; increment){
   code to be executed;
}
```

while loop

Syntax

```
while (condition) {
   code to be executed;
}
```

do-while loop

```
do {
   code to be executed;
}
while (condition);
```

f**oreach**

```
foreach (array as value) {
   code to be executed;
}
```

Arrays

array is an ordered map (contains value on the basis of key). It is used to hold multiple values of similar type in a single variable.

Indexed Array

index is represented by number which starts from 0. We can store number, string and object in the PHP array. All PHP array elements are assigned to an index number by default.

to define indexed array:

```
$season=array("summer","winter","spring","autumn");
```

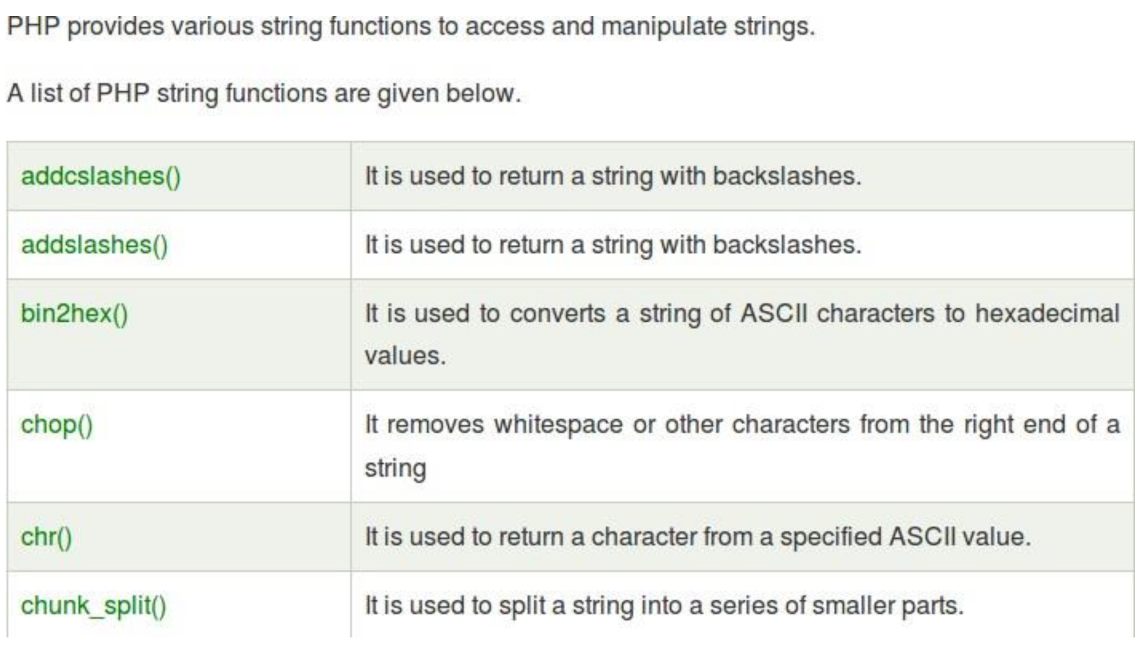
Associative Array

We can associate name with each array elements in PHP using => symbol.

to define associative array:

```
$salary=array("Sonoo"=>"350000","John"=>"450000","Kartik"=>"200000");
```

String Functions



PHP provides various string functions to access and manipulate strings.

A list of PHP string functions are given below.

addcslashes()	It is used to return a string with backslashes.
addslashes()	It is used to return a string with backslashes.
bin2hex()	It is used to converts a string of ASCII characters to hexadecimal values.
chop()	It removes whitespace or other characters from the right end of a string
chr()	It is used to return a character from a specified ASCII value.
chunk_split()	It is used to split a string into a series of smaller parts.

Functions

function is a piece of code that can be reused many times. It can take input as argument list and return value. There are thousands of built-in functions in PHP.

User-defined Functions

We can declare and call user-defined functions easily. Let's see the syntax to declare user-defined functions.

Syntax

```
1. function functionname()
2. {
3. //code to be executed
4. }
```

```
<?php
function sayHello(){
echo "Hello PHP Function";
}
sayHello();//calling function
?>
```

o/p:Hello PHP Function

Call by value

In call by value, actual value is not modified if it is modified inside the function.

```
<?php
        function increment($i)
        {
           $i++;
        }
        $i = 10;
        increment($i);
        echo $i;
        ?>
```

o/p:10

Call By Reference

Value passed to the function doesn't modify the actual value by default (call by value). But we can do so by passing value as a reference.

By default, value passed to the function is call by value. To pass value as a reference, you need to use ampersand (&) symbol before the argument name.

```
<?php
        function adder(&$str2)
        {
           $str2 .= 'Call By Reference';
        }
        $str = 'Hello ';
        adder($str);
        echo $str;
        ?>
```

```
Hello Call By Reference
```

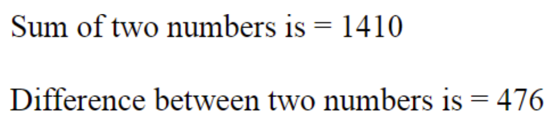
Parameterized Function

PHP Parameterized functions are the functions with parameters. You can pass any number of parameters inside a function. These passed parameters act as variables inside your function.

They are specified inside the parentheses, after the function name.

The output depends upon the dynamic values passed as the parameters into the function.

```
<!DOCTYPE html>
        <html>
        <head>
          <title>Parameter Addition and Subtraction Example</title>
        </head>
        <body>
        <?php
             //Adding two numbers
           function add($x, $y) {
              $sum = $x + $y;
             echo "Sum of two numbers is = $sum <br><br>";
           }
           add(467, 943);
      //Subtracting two numbers
             function sub($x, $y) {
               $diff = $x - $y;
              echo "Difference between two numbers is = $diff";
            }
          sub(943, 467);
         ?>
        </body>
      </html>
```



```
Sum of two numbers is = 1410

Difference between two numbers is = 476
```

File Handling

PHP File System allows us to create file, read file line by line, read file character by character, write file, append file, delete file and close file.

Open File - fopen()

fopen() function is used to open a file.

Syntax

```
resource fopen ( string $filename , string $mode [, bool $use_include_path = fa
lse [, resource $context
```

```
<?php
        $handle = fopen("c:\\folder\\file.txt", "r");
        ?>
```

Close File - fclose()

fclose() function is used to close an open file pointer.

Syntax

```
ool fclose ( resource $handle )
```

Example

```
<?php
fclose($handle);
?>
```

Read File - fread()

The PHP fread() function is used to read the content of the file. It accepts two arguments: resource and file size.

Syntax

```
string fread ( resource $handle , int $length )
<?php
$filename = "c:\\myfile.txt";
$handle = fopen($filename, "r");//open file in read mode

$contents = fread($handle, filesize($filename));//read file

echo $contents;//printing data of file
fclose($handle);//close file
?>
```

Output

```
hello php file
```

Write File - fwrite()

fwrite() function is used to write content of the string into file

Syntax

```
int fwrite ( resource $handle , string $string [, int $length ] )
```

Example

```
<?php
$fp = fopen('data.txt', 'w');//open file in write mode
fwrite($fp, 'hello ');
fwrite($fp, 'php file');
fclose($fp);

echo "File written successfully";
?>
```

Output

```
File written successfully
```

Form Handling

We can create and use forms in PHP. To get form data, we need to use PHP superglobals $_GET and $_POST.

The form request may be get or post. To retrieve data from get request, we need to use $_GET, for post request $_POST.

Get Form

Get request is the default form request. The data passed through get request is visible on the URL browser so it is not secured. You can send limited amount of data through get request.

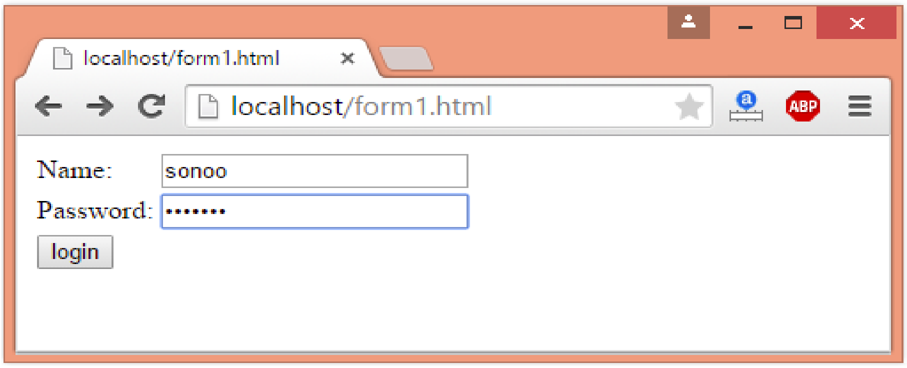
```
<form action="welcome.php" method="get">
        Name: <input type="text" name="name"/>
        <input type="submit" value="visit"/>
        </form>
<?php
        $name=$_GET["name"];//receiving name field value in $name variable
        echo "Welcome, $name";
        ?>
```

Post Form

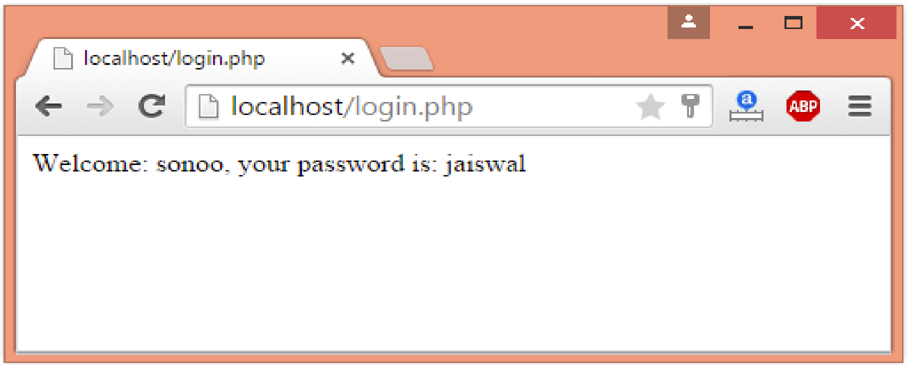
Post request is widely used to submit form that have large amount of data such as file upload, image upload, login form, registration form etc.

The data passed through post request is not visible on the URL browser so it is secured. You can send large amount of data through post request.

```
<form action="login.php" method="post">
        <table>
        <tr><td>Name:</td><td> <input type="text" name="name"/></td></tr>
        <tr><td>Password:</td><td> <input type="password" name="password"/></td
        ></tr>
        <tr><td colspan="2"><input type="submit" value="login"/>  </td></tr>
        </table>
        </form>
    <?php
        $name=$_POST["name"];//receiving name field value in $name variable
        $password=$_POST["password"];//receiving password field value in $passwor
        d variable
        echo "Welcome: $name, your password is: $password";
        ?>
```



```
sonoo
•••••••
```



```
Welcome: sonoo, your password is: jaiswal
```

UNIT- V
Ruby : Introduction to Ruby, Feature of Ruby, Data types, Variables, Operators, Conditional statements, Loops, , Arrays, Strings, Hashes, working on Methods, Blocks, and Modules.

Ruby is a object-oriented, reflective, general-purpose, dynamic programming language. Ruby was developed to make it act as a sensible buffer between human programmers and the underlying computing machinery. It is an interpreted scripting language which means most of its implementations execute instructions directly and freely, without previously compiling a program into machine-language instructions. Ruby is used to create web applications of different sorts. It is one of the hot technology at present to create web applications.

Data types

Data types represents a type of data such as text, string, numbers, etc. There are different data types in Ruby:

- Numbers
- Strings
- Symbols
- Hashes
- Arrays

Numbers

Integers and floating point numbers come in the category of numbers.

Integers are held internally in binary form. Integer numbers are numbers without a fraction. According to their size, there are two types of integers. One is Bignum and other is Fixnum.

Strings

A string is a group of letters that represent a sentence or a word. Strings are defined by enclosing a text within single (') or double (") quote.

Symbols

Symbols are like strings. A symbol is preceded by a colon (:). For example,

1. :abcd

They do not contain spaces. Symbols containing multiple words are written with (_). One difference between string and symbol is that, if text is a data then it is a string but if it is a code it is a symbol.

Symbols are unique identifiers and represent static values, while string represent values that change.

Hashes

A hash assign its values to its keys. They can be looked up by their keys. Value to a key is assigned by => sign. A key/value pair is separated with a comma between them and all the pairs are enclosed within curly braces. For example,

- {"Akash" => "Physics", "Ankit" => "Chemistry", "Aman" => "Maths"}

Arrays

An array stroes data or list of data. It can contain all types of data. Data in an array are separated by comma in between them and are enclosed by square bracket. For example,

1. ["Akash", "Ankit", "Aman"]

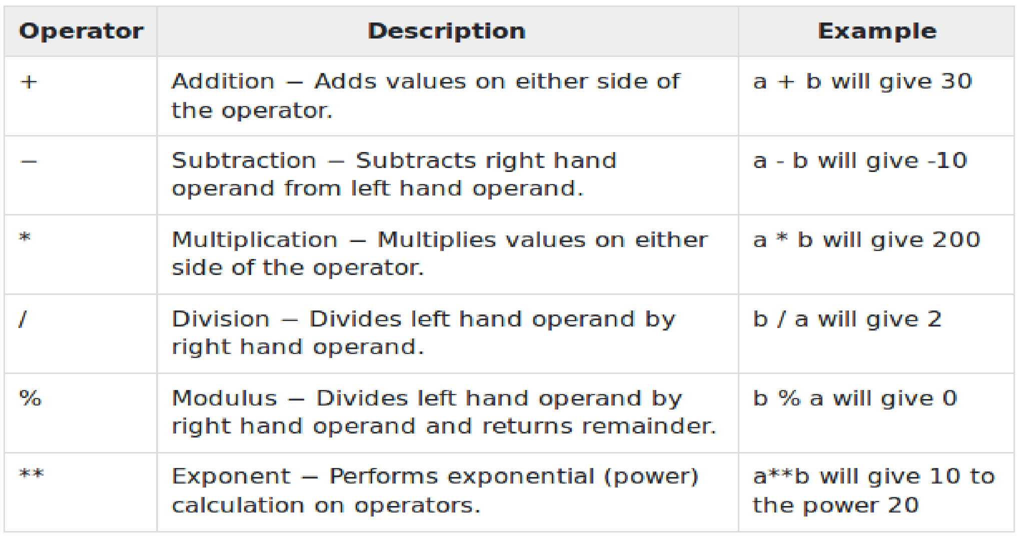
Variables

Variables are the memory locations, which hold any data to be used by any program. Local variables begin with a lowercase letter or _. The scope of a local variable ranges from class, module, def, or do to the corresponding end or from a block's opening brace to its close brace {}.When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.Global variables begin with $. Uninitialized global variables have the value nil and produce warnings with the -w option.Assignment to global variables alters the global status. It is not recommended to use global variables. They make programs cryptic.

Operators

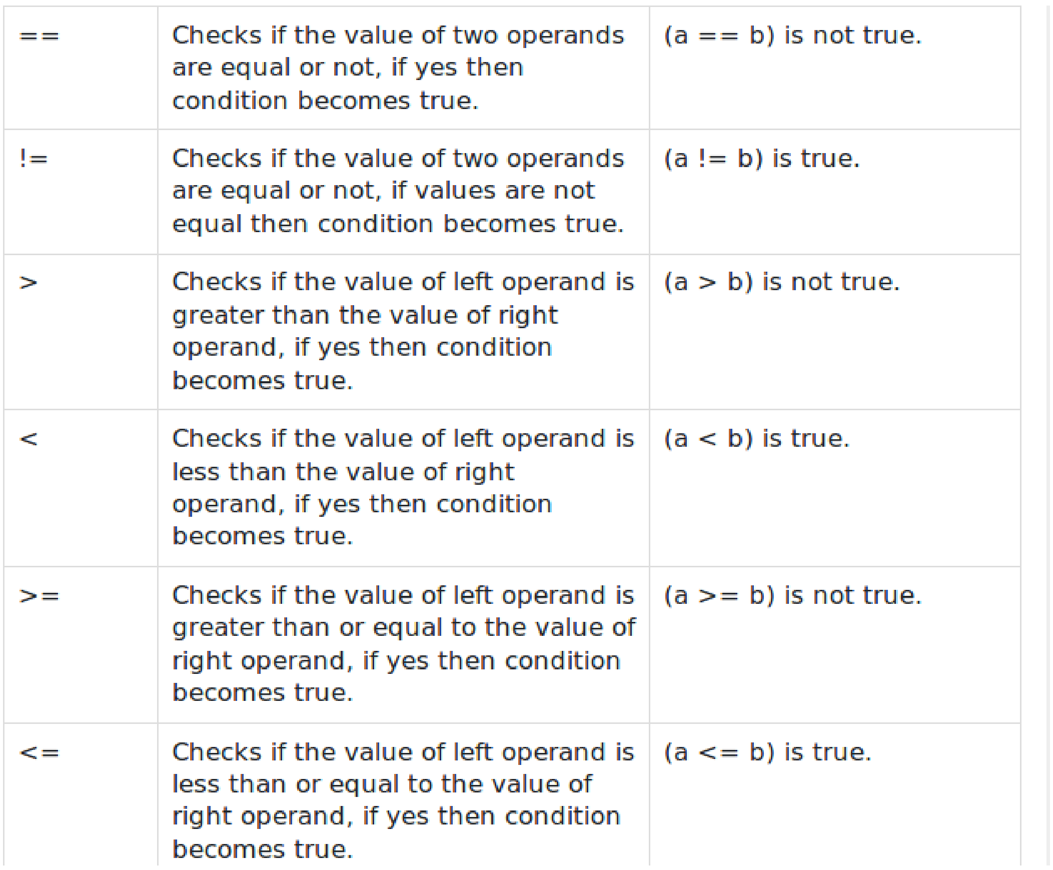
Ruby supports a rich set of operators, as you'd expect from a modern language. Most operators are actually method calls. For example, a + b is interpreted as a.+(b), where the + method in the object referred to by variable a is called with b as its argument.

Arithmetic Operators



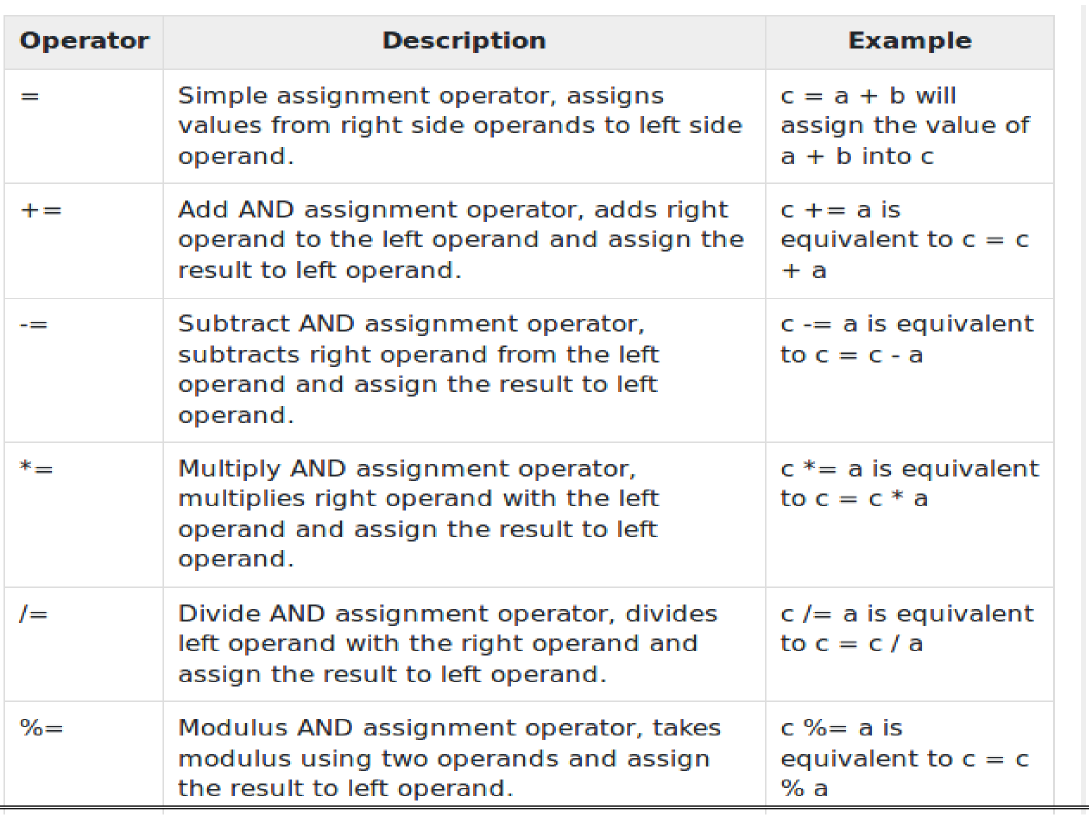
Operator	Description	Example
+	Addition − Adds values on either side of the operator.	a + b will give 30
−	Subtraction − Subtracts right hand operand from left hand operand.	a - b will give -10
*	Multiplication − Multiplies values on either side of the operator.	a * b will give 200
/	Division − Divides 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
**	Exponent − Performs exponential (power) calculation on operators.	a**b will give 10 to the power 20

Comparison Operators



==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	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.
<	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.
>=	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.
<=	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.

Assignment Operators



Operator	Description	Example
=	Simple assignment operator, assigns values from right side operands to left side operand.	c = a + b will assign the value of a + b into c
+=	Add AND assignment operator, 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, 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, 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, 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, takes modulus using two operands and assign the result to left operand.	c %= a is equivalent to c = c % a

Bitwise Operators

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
```

Logical Operators

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true, then the condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a or b) is true.
&&	Called Logical AND operator. If both the operands are non zero, then the condition becomes true.	(a && b) is true.
||	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a || b) is true.
!	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 false.
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.	not(a && b) is false.

Ternary Operator

It 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.

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Conditional statements

Ruby offers conditional structures that are pretty common to modern languages.

Ruby if...else Statement

Syntax

```
if conditional [then]
  code...
[elsif conditional [then]
  code...]...
[else
  code...]
end
#!/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 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.

```
#!/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
```

Loops

Loops in Ruby are used to execute the same block of code a specified number of times.

Ruby while Statement

Syntax

```
while conditional [do]
   code
end
```

```
#!/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
```

for statement

Syntax

```
for variable [, variable ...] in expression [do]
   code
end
```

```
#!/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
```

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.

```
def method_name
   expr..
end
```

```
#!/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 Statement

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

```
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
```

Blocks

A block consists of chunks of code.

- You assign a name to a block.
- The code in the block is always enclosed within braces ({}).
- A block is always invoked from a function with the same name as that of the block. This means that if you have a block with the name test, then you use the function test to invoke this block.
- You invoke a block by using the yield statement.

syntax

```
block_name {
  statement1
  statement2
  ..........
}
```

```
#!/usr/bin/ruby
def test
  puts "You are in the method"
  yield
  puts "You are again back to the method"
  yield
end
test {puts "You are in the block"}
```

This will produce the following result −

```
You are in the method
You are in the block
You are again back to the method
You are in the block
```

Modules

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits.

- Modules provide a namespace and prevent name clashes.
- Modules implement the mixin facility.

Modules define a namespace, a sandbox in which your methods and constants can play without having to worry about being stepped on by other methods and constants.

Syntax

```
module Identifier
   statement1
   statement2
   ...........
end
```

Module constants are named just like class constants, with an initial uppercase letter. The method definitions look similar, too: Module methods are defined just like class methods.

As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

Example

```
#!/usr/bin/ruby
# Module defined in trig.rb file
module Trig
   PI = 3.141592654
   def Trig.sin(x)
   # ..
   end
   def Trig.cos(x)
   # ..
   end
end
```

We can define one more module with the same function name but different functionality

```
#!/usr/bin/ruby
# Module defined in moral.rb file
module Moral
   VERY_BAD = 0
   BAD = 1
   def Moral.sin(badness)
   # ...
   end
end
```

Like class methods, whenever you define a method in a module, you specify the module name followed by a dot and then the method name.

Arrays

Ruby arrays are ordered, integer-indexed collections of any object. Each element in an array is associated with and referred to by an index.

Array indexing starts at 0, as in C or Java. A negative index is assumed relative to the end of the array --- that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.

Creating Arrays

There are many ways to create or initialize an array. One way is with the new class method −

```
names = Array.new
```

You can set the size of an array at the time of creating array −

```
names = Array.new(20)
```

The array names now has a size or length of 20 elements.

```
#!/usr/bin/ruby
names = Array.new(20)
puts names.size # This returns 20
puts names.length # This also returns 20
```

This will produce the following result −

```
20
20
```

You can assign a value to each element in the array as follows

```
#!/usr/bin/ruby
names = Array.new(4, "mac")
puts "#{names}"
```

This will produce the following result −

```
["mac", "mac", "mac", "mac"]
```

You can also use a block with new, populating each element with what the block evaluates to

```
#!/usr/bin/ruby
nums = Array.new(10) { |e| e = e * 2 }
puts "#{nums}"
```

This will produce the following result −

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

There is another method of Array, []. It works like this −

```
nums = Array.[](1, 2, 3, 4,5)
```

One more form of array creation is as follows −

```
nums = Array[1, 2, 3, 4,5]
```

Strings

A String object in Ruby holds and manipulates an arbitrary sequence of one or more bytes, typically representing characters that represent human language.

The simplest string literals are enclosed in single quotes (the apostrophe character). The text within the quote marks is the value of the string −

```
'This is a simple Ruby string literal'
```

Expression Substitution

Expression substitution is a means of embedding the value of any Ruby expression into a string using #{ and } −

```
#!/usr/bin/ruby
x, y, z = 12, 36, 72
puts "The value of x is #{ x }."
puts "The sum of x and y is #{ x + y }."
puts "The average was #{ (x + y + z)/3 }."
```

This will produce the following result −

```
The value of x is 12.
The sum of x and y is 48.
The average was 40.
```

We need to have an instance of String object to call a String method. Following is the way to create an instance of String object −

```
new [String.new(str = "")]
```

This will return a new string object containing a copy of str. Now, using str object, we can all use any available instance methods.

```
#!/usr/bin/ruby
myStr = String.new("THIS IS TEST")
foo = myStr.downcase
puts "#{foo}"
```

This will produce the following result −

```
this is test
```

Hashes

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
#!/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
```

File I/O

Ruby provides a whole set of I/O-related methods implemented in the Kernel module. All the I/O methods are derived from the class IO.

The class IO provides all the basic methods, such as read, write, gets, puts, readline, getc, and printf.

The puts Statement

In the previous chapters, you have assigned values to variables and then printed the output using puts statement.

The puts statement instructs the program to display the value stored in the variable. This will add a new line at the end of each line it writes.

```
#!/usr/bin/ruby
val1 = "This is variable one"
val2 = "This is variable two"
puts val1
puts val2
```

This will produce the following result −

```
This is variable one
This is variable two
```

The gets Statement

The gets statement can be used to take any input from the user from standard screen called STDIN.

Example

The following code shows you how to use the gets statement. This code will prompt the user to enter a value, which will be stored in a variable val and finally will be printed on STDOUT.

```
#!/usr/bin/ruby
puts "Enter a value :"
val = gets
puts val
```

This will produce the following result −

```
Enter a value :
This is entered value
This is entered value
```

The putc Statement

Unlike the puts statement, which outputs the entire string onto the screen, the putc statement can be used to output one character at a time.

Example

The output of the following code is just the character H −

```
#!/usr/bin/ruby
str = "Hello Ruby!"
putc str
```

This will produce the following result −

```
H
```

The print Statement

The print statement is similar to the puts statement. The only difference is that the puts statement goes to the next line after printing the contents, whereas with the print statement the cursor is positioned on the same line.

```
#!/usr/bin/ruby
print "Hello World"
print "Good Morning"
```

This will produce the following result −

```
Hello WorldGood Morning
```

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to play with actual data files.

The File.new Method

You can create a File object using File.new method for reading, writing, or both, according to the mode string. Finally, you can use File.close method to close that file.

Syntax

```
aFile = File.new("filename", "mode")
```

```
   # ... process the file
aFile.close
```

The File.open Method

You can use File.open method to create a new file object and assign that file object to a file. However, there is one difference in between File.open and File.new methods. The difference is that the File.open method can be associated with a block, whereas you cannot do the same using the File.new method.

```
File.open("filename", "mode") do |aFile|
   # ... process the file
end
```

Here is a list of The Different Modes of opening a File −

Reading and Writing Files

The same methods that we've been using for 'simple' I/O are available for all file objects. So, gets reads a line from standard input, and aFile.gets reads a line from the file object aFile.

The sysread Method

You can use the method sysread to read the contents of a file. You can open the file in any of the modes when using the method sysread. For example −

Following is the input text file −

This is a simple text file for testing purpose.

Now let's try to read this file −

```
#!/usr/bin/ruby
aFile = File.new("input.txt", "r")
if aFile
   content = aFile.sysread(20)
   puts content
else
   puts "Unable to open file!"
end
```

This statement will output the first 20 characters of the file. The file pointer will now be placed at the 21st character in the file.

The syswrite Method

You can use the method syswrite to write the contents into a file. You need to open the file in write mode when using the method syswrite. For example −

```
#!/usr/bin/ruby
aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
else
   puts "Unable to open file!"
```

end

Renaming and Deleting Files

You can rename and delete files programmatically with Ruby with the rename and delete methods.

Following is the example to rename an existing file test1.txt −

```
#!/usr/bin/ruby
# Rename a file from test1.txt to test2.txt
File.rename( "test1.txt", "test2.txt" )
```

Following is the example to delete an existing file test2.txt −

```
#!/usr/bin/ruby
# Delete file test2.txt
File.delete("test2.txt")
```

Form Handling

Form

To create a form tag with the specified action, and with POST request, use the following syntax −

```
<%= form_tag :action => 'update', :id => @some_object %>

<%= form_tag( { :action => :save, }, { :method => :post }) %>
```

Use :multipart => true to define a MIME-multipart form (for file uploads).

```
<%= form_tag( {:action => 'upload'}, :multipart => true ) %>
```

File Upload

Define a multipart form in your view −

```
<%= form_tag( { :action => 'upload' }, :multipart => true ) %>
   Upload file: <%= file_field( "form", "file" ) %>
   <br />

   <%= submit_tag( "Upload file" ) %>
<%= end_form_tag %>
```

Handle the upload in the controller −

```
def upload
   file_field = @params['form']['file'] rescue nil

   # file_field is a StringIO object
   file_field.content_type # 'text/csv'
   file_field.full_original_filename
   ...
end
```

Text Fields

To create a text field use the following syntax −

```
<%= text_field :modelname, :attribute_name, options %>
```

Have a look at the following example −

```
<%= text_field "person", "name", "size" => 20 %>
```

This will generate following code −

```
<input type = "text" id = "person_name" name = "person[name]"
   size = "20" value = "<%= @person.name %>" />
```

To create hidden fields, use the following syntax;

```
<%= hidden_field ... %>
```

To create password fields, use the following syntax;

```
<%= password_field ... %>
```

To create file upload fields, use the following syntax;

```
<%= file_field ... %>
```

Text Area

To create a text area, use the following syntax −

```
<%= text_area ... %>
```

Have a look at the following example −

```
<%= text_area "post", "body", "cols" => 20, "rows" => 40%>
```

This will generate the following code −

```
<textarea cols = "20" rows = "40" id = "post_body" name =" post[body]">
   <%={@post.body}%>
</textarea>
```

Radio Button

To create a Radio Button, use the following syntax −

```
<%= radio_button :modelname, :attribute, :tag_value, options %>
```

Have a look at the following example −

```
radio_button("post", "category", "rails")
radio_button("post", "category", "java")
```

This will generate the following code −

```
<input type = "radio" id = "post_category" name = "post[category]"
   value = "rails" checked = "checked" />
<input type = "radio" id = "post_category" name = "post[category]" value = "java" />
```

Checkbox Button

To create a Checkbox Button use the following syntax −

```
<%= check_box :modelname, :attribute,options,on_value,off_value%>
```

Have a look at the following example −

```
check_box("post", "validated")
```

This will generate the following code −

```
<input type = "checkbox" id = "post_validate" name = "post[validated]"
   value = "1" checked = "checked" />
<input name = "post[validated]" type = "hidden" value = "0" />
```

Let's check another example −

```
check_box("puppy", "gooddog", {}, "yes", "no")
```

This will generate following code −

```
<input type = "checkbox" id = "puppy_gooddog" name = "puppy[gooddog]" value =
"yes" />
<input name = "puppy[gooddog]" type = "hidden" value = "no" />
```

Options

To create a dropdopwn list, use the following syntax −

```
<%= select :variable,:attribute,choices,options,html_options%>
```

Have a look at the following example −

```
select("post", "person_id", Person.find(:all).collect {|p| [ p.name, p.id ] })
```

This could generate the following code. It depends on what value is available in your database. −

```
<select name = "post[person_id]">
   <option value = "1">David</option>
   <option value = "2">Sam</option>
   <option value = "3">Tobias</option>
</select>
```

Date Time

Following is the syntax to use data and time −

```
<%= date_select :variable, :attribute, options %>
<%= datetime_select :variable, :attribute, options %>
```

Following are examples of usage −

```
<%=date_select "post", "written_on"%>
<%=date_select "user", "birthday", :start_year => 1910%>
<%=date_select "user", "cc_date", :start_year => 2005,
   :use_month_numbers => true, :discard_day => true, :order => [:year, :month]%>
<%=datetime_select "post", "written_on"%>
```

End Form Tag

Use following syntax to create </form> tag −

```
<%= end_form_tag %>
```