# Dr. SNS RAJALAKSHMI COLLEGE OF ARTS & SCIENCE (AUTONOMOUS)

Accredited by NAAC (Cycle III) with 'A+' Grade

Affiliated to Bharathiar University

Coimbatore-641049

**DEPARTMENT OF COMPUTER APPLICATIONS**

**III BCA**

**WEB DEDIGNING-16UCA504**

**PREPARED BY: Mrs.Lalitha**

UNIT-IV

# JavaScript

JavaScript is the world's most popular programming language.

JavaScript is the programming language of the Web.

JavaScript is easy to learn.

JavaScript Can Change HTML Content

One of many JavaScript HTML methods is getElementById().

The example below "finds" an HTML element (with id="demo"), and changes the element content (innerHTML) to "Hello JavaScript":

Example

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

JavaScript Can Change HTML Attribute Values

In this example JavaScript changes the value of the src (source) attribute of an <img> tag:

JavaScript Can Change HTML Styles (CSS)

Changing the style of an HTML element, is a variant of changing an HTML attribute:

Example

```
document.getElementById("demo").style.fontSize = "35px";
```

JavaScript Can Hide HTML Elements

Hiding HTML elements can be done by changing the display style:

Example

```
document.getElementById("demo").style.display = "none";
```

JavaScript Can Show HTML Elements

Showing hidden HTML elements can also be done by changing the display style:

Example

```
document.getElementById("demo").style.display = "block";
```

The HTML DOM Document Object

The document object represents your web page.

If you want to access any element in an HTML page, you always start with accessing the document object.

Below are some examples of how you can use the document object to access and manipulate HTML.

Finding HTML Elements

| **Method** | **Description** |
|---|---|
| document.getElementById(*id*) | Find an element by element id |
| document.getElementsByTagName(*name*) | Find elements by tag name |
| document.getElementsByClassName(*name*) | Find elements by class name |

Changing HTML Elements

| **Property** | **Description** |
|---|---|
| *element*.innerHTML = *new html content* | Change the inner HTML of an element |
| *element.attribute = new value* | Change the attribute value of an HTML elem |
| *element*.style.*property = new style* | Change the style of an HTML element |

| Method | Description |
| --- | --- |
| *element*.setAttribute*(attribute, value)* | Change the attribute value of an HTML elem |

Adding and Deleting Elements

| Method | Description |
| --- | --- |
| document.createElement(*element*) | Create an HTML element |
| document.removeChild(*element*) | Remove an HTML element |
| document.appendChild(*element*) | Add an HTML element |
| document.replaceChild(*new, old*) | Replace an HTML element |
| document.write(*text*) | Write into the HTML output stream |

---

---

Adding Events Handlers

| Method | Description |
| --- | --- |
| document.getElementById(*id*).onclick = function(){*code*} | Adding event handler code to an onclick even |

Finding HTML Objects

The first HTML DOM Level 1 (1998), defined 11 HTML objects, object collections, and properties. These are still valid in HTML5.

Later, in HTML DOM Level 3, more objects, collections, and properties were added.

Purpose of DOM:

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

It is a programming interface that allows us to create, change, or remove elements from a website document. DOM manipulation is when you use JavaScript **to add, remove, and modify elements of a website**. It is very common in web development.

JavaScript Function Definitions

JavaScript functions are **defined** with the function keyword.

You can use a function **declaration** or a function **expression**.

Function Declarations

Earlier in this tutorial, you learned that functions are **declared** with the following syntax:

```
function functionName(parameters) {
  // code to be executed
}
```

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

Example

```
function myFunction(a, b) {
  return a * b;
}
```

Semicolons are used to separate executable JavaScript statements.
Since a function **declaration** is not an executable statement, it is not common to end it with a semicolon.

Function Expressions

A JavaScript function can also be defined using an **expression**.

A function expression can be stored in a variable:

Example

```
const x = function (a, b) {return a * b};
```

After a function expression has been stored in a variable, the variable can be used as a function:

Example

```
const x = function (a, b) {return a * b};
let z = x(4, 3);
```

The function above is actually an **anonymous function** (a function without a name).

Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

The function above ends with a semicolon because it is a part of an executable statement.

The Function() Constructor

As you have seen in the previous examples, JavaScript functions are defined with the function keyword.

Functions can also be defined with a built-in JavaScript function constructor called Function().

Example

```
const myFunction = new Function("a", "b", "return a * b");

let x = myFunction(4, 3);
```

You actually don't have to use the function constructor. The example above is the same as writing:

Example

```
const myFunction = function (a, b) {return a * b};

let x = myFunction(4, 3);
```

Most of the time, you can avoid using the new keyword in JavaScript.

Function Hoisting

Earlier in this tutorial, you learned about "hoisting" (JavaScript Hoisting).

Hoisting is JavaScript's default behavior of moving **declarations** to the top of the current scope.

Hoisting applies to variable declarations and to function declarations.

Because of this, JavaScript functions can be called before they are declared:

```
myFunction(5);

function myFunction(y) {
  return y * y;
}
```

Functions defined using an expression are not hoisted.

Self-Invoking Functions

Function expressions can be made "self-invoking".

A self-invoking expression is invoked (started) automatically, without being called.

Function expressions will execute automatically if the expression is followed by ().

You cannot self-invoke a function declaration.

You have to add parentheses around the function to indicate that it is a function expression:

Example

```
(function () {
  let x = "Hello!!";  // I will invoke myself
})();
```

The function above is actually an **anonymous self-invoking function** (function without name).

Functions Can Be Used as Values

JavaScript functions can be used as values:

Example

```
function myFunction(a, b) {
  return a * b;
}

let x = myFunction(4, 3);
```

JavaScript functions can be used in expressions:

Example

```
function myFunction(a, b) {
  return a * b;
}

let x = myFunction(4, 3) * 2;
```

Functions are Objects

The typeof operator in JavaScript returns "function" for functions.

But, JavaScript functions can best be described as objects.

JavaScript functions have both **properties** and **methods**.

The arguments.length property returns the number of arguments received when the function was invoked:

Example

```
function myFunction(a, b) {
  return arguments.length;
}
```

The toString() method returns the function as a string:

Example

```
function myFunction(a, b) {
  return a * b;
}

let text = myFunction.toString();
```

A function defined as the property of an object, is called a method to the object.
A function designed to create new objects, is called an object constructor.

Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

Example

```
// ES5
var x = function(x, y) {
  return x * y;
}

// ES6
const x = (x, y) => x * y;
```

Arrow functions do not have their own this. They are not well suited for defining **object methods**.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using const is safer than using var, because a function expression is always constant value.

You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

Example

```
const x = (x, y) => { return x * y };
```

JavaScript Form :

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:

JavaScript Example

```
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("Name must be filled out");
    return false;
  }
}
```

The function can be called when the form is submitted:

HTML Form Example

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
Name: <input type="text" name="fname">
<input type="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

Submit

Automatic HTML Form Validation

HTML form validation can be performed automatically by the browser:

If a form field (fname) is empty, the required attribute prevents this form from being submitted:

HTML Form Example

```
<form action="/action_page.php" method="post">
  <input type="text" name="fname" required>
  <input type="submit" value="Submit">
</form>
```

Data Validation

Data validation is the process of ensuring that user input is clean, correct, and useful.

Typical validation tasks are:

- has the user filled in all required fields?
- has the user entered a valid date?
- has the user entered text in a numeric field?

Most often, the purpose of data validation is to ensure correct user input.

Validation can be defined by many different methods, and deployed in many different ways.

**Server side validation** is performed by a web server, after input has been sent to the server.

**Client side validation** is performed by a web browser, before input is sent to a web server.

---

HTML Constraint Validation

HTML5 introduced a new HTML validation concept called **constraint validation**.

HTML constraint validation is based on:

- Constraint validation **HTML Input Attributes**
- Constraint validation **CSS Pseudo Selectors**
- Constraint validation **DOM Properties and Methods**

Constraint Validation HTML Input Attributes

| Attribute | Description |
|---|---|
| Disabled | Specifies that the input element should be disabled |
| Max | Specifies the maximum value of an input element |

| | |
|---|---|
| Min | Specifies the minimum value of an input element |
| Pattern | Specifies the value pattern of an input element |
| Required | Specifies that the input field requires an element |
| type | Specifies the type of an input element |

Using a JavaScript Library

Change this:

```
<script src=""></script>
```

To this:

```
<script src="https://www.w3schools.com/lib/w3.js"></script>
```

How to - Slideshow

JavaScript Example

```
<img class="nature" src="img_snowtops.jpg" width="100%">
<img class="nature" src="img_mountains.jpg" width="100%">
<img class="nature" src="img_nature.jpg" width="100%">

<script>
w3.slideshow(".nature", 1500);
</script>
```

Hide/Show HTML Elements

JavaScript Example

```
<p>
<button onclick="w3.hide('#London')">Hide London</button>
<button onclick="w3.show('#London')">Show London</button>
</p>
```

```
<div id="London" class="w3-container w3-red">
  <h2>London</h2>
  <p>London is the capital city of England.</p>
</div>

<div id="Paris" class="w3-container w3-green">
  <h2>Paris</h2>
  <p>Paris is the capital of France.</p>
</div>

<div id="Tokyo" class="w3-container w3-blue">
  <h2>Tokyo</h2>
  <p>Tokyo is the capital of Japan.</p>
</div>
```

JavaScript control statement:

JavaScript statements are the commands to tell the browser to what action to perform. Statements are separated by semicolon (;).

JavaScript statement constitutes the JavaScript code which is translated by the browser line by line.

Example of JavaScript statement:

```
document.getElementById("demo").innerHTML = "Welcome";
```

Following table shows the various JavaScript Statements −

| Sr.No. | Statement | Description |
|---|---|---|
| 1. | switch case | A block of statements in which execution of code depends upon different cases. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a **default** condition will be used. |
| 2. | If else | The **if** statement is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally. |
| 3. | While | The purpose of a while loop is to execute a statement or code block repeatedly as long as expression is true. Once expression becomes false, the loop will be exited. |

| | | |
|---|---|---|
| 4. | do while | Block of statements that are executed at least once and continues to be executed while condition is true. |
| 5. | for | Same as while but initialization, condition and increment/decrement is done in the same line. |
| 6. | for in | This loop is used to loop through an object's properties. |
| 7. | continue | The continue statement tells the interpreter to immediately start the next iteration of the loop and skip remaining code block. |
| 8. | break | The break statement is used to exit a loop early, breaking out of the enclosing curly braces. |
| 9. | function | A function is a group of reusable code which can be called anywhere in your programme. The keyword function is used to declare a function. |
| 10. | return | Return statement is used to return a value from a function. |
| 11. | var | Used to declare a variable. |
| 12. | try | A block of statements on which error handling is implemented. |
| 13. | catch | A block of statements that are executed when an error occur. |
| 14. | throw | Used to throw an error. |

JavaScript Comments

JavaScript supports both C-style and C++-style comments, thus:

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

Example

```
<script language="javascript" type="text/javascript">
  <!--

    // this is a comment. It is similar to comments in C++

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

Javascript Operators

JavaScript includes operators same as other languages. An operator performs some operation on single or multiple operands (data value) and produces a result. For example, in `1 + 2`, the `+` sign is an operator and 1 is left side operand and 2 is right side operand. The `+` operator performs the addition of two numeric values and returns a result.

JavaScript includes following categories of operators.

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Assignment Operators
5. Conditional Operators
6. Ternary Operator

## Arithmetic Operators

Arithmetic operators are used to perform mathematical operations between numeric operands.

| Operator | Description |
|---|---|
| + | Adds two numeric operands. |
| - | Subtract right operand from left operand |
| * | Multiply two numeric operands. |
| / | Divide left operand by right operand. |
| % | Modulus operator. Returns remainder of two operands. |
| ++ | Increment operator. Increase operand value by one. |
| -- | Decrement operator. Decrease value by one. |

The following example demonstrates how arithmetic operators perform different tasks on operands.

### Example: Arithmetic Operation

```
let x = 5, y = 10;

let z = x + y; //performs addition and returns 15

z = y - x; //performs subtraction and returns 5

z = x * y; //performs multiplication and returns 50

z = y / x; //performs division and returns 2

z = x % 2; //returns division remainder 1
```

The `++` and `--` operators are unary operators. It works with either left or right operand only. When used with the left operand, e.g., `x++`, it will increase the value of `x` when the program control goes to the next statement. In the same way, when it is used with the right operand, e.g., `++x`, it will increase the value of `x` there only. Therefore, `x++` is called post-increment, and `++x` is called pre-increment.

### Example: Post and Pre Increment/Decrement

```
let x = 5;

x++; //post-increment, x will be 5 here and 6 in the next line

++x; //pre-increment, x will be 7 here
```

```
x--; //post-decrement, x will be 7 here and 6 in the next line

--x; //pre-decrement, x will be 5 here
```

### String Concatenation

The + operator performs concatenation operation when one of the operands is of string type. The following example demonstrates string concatenation even if one of the operands is a string.

#### Example: + Operator with String

```
let a = 5, b = "Hello ", c = "World!", d = 10;

a + b; //returns "5Hello "

b + c; //returns "Hello World!"

a + d; //returns 15

b + true; //returns "Hello true"

c - b; //returns NaN; - operator can only used with numbers
```

### Comparison Operators

JavaScript provides comparison operators that compare two operands and return a boolean value true or false.

| Operators | Description |
|---|---|
| == | Compares the equality of two operands without considering type. |
| === | Compares equality of two operands with type. |
| != | Compares inequality of two operands. |
| > | Returns a boolean value true if the left-side value is greater than the right-side value; otherwise, returns false. |
| < | Returns a boolean value true if the left-side value is less than the right-side value; otherwise, returns false. |
| >= | Returns a boolean value true if the left-side value is greater than or equal to the right-side value; otherwise, returns false. |

| Operators | Description |
|---|---|
| <= | Returns a boolean value true if the left-side value is less than or equal to the right-side value; otherwise, returns false. |

The following example demonstrates the comparison operators.

Example: JavaScript Comparison Operators

```
let a = 5, b = 10, c = "5";
let x = a;

a == c; // returns true

a === c; // returns false

a == x; // returns true

a != b; // returns true

a > b; // returns false

a < b; // returns true

a >= b; // returns false

a <= b; // returns true
```

## Logical Operators

In JavaScript, the logical operators are used to combine two or more conditions. JavaScript provides the following logical operators.

| Operator | Description |
|---|---|
| && | && is known as AND operator. It checks whether two operands are non-zero or not (0, false, undefined, null or "" are considered as zero). It returns 1 if they are non-zero; otherwise, returns 0. |
| \|\| | \|\| is known as OR operator. It checks whether any one of the two operands is non-zero or not (0, false, undefined, null or "" is considered as zero). It returns 1 if any one of of them is non-zero; otherwise, returns 0. |
| ! | ! is known as NOT operator. It reverses the boolean result of the operand (or |

| Operator | Description |
| --- | --- |
| | condition). !false returns true, and !true returns false. |

Example: Logical Operators

```
let a = 5, b = 10;

(a != b) && (a < b); // returns true

(a > b) || (a == b); // returns false

(a < b) || (a == b); // returns true

!(a < b); // returns false

!(a > b); // returns true
```

### Assignment Operators

JavaScript provides the assignment operators to assign values to variables with less key strokes.

| Assignment operators | Description |
| --- | --- |
| = | Assigns right operand value to the left operand. |
| += | Sums up left and right operand values and assigns the result to the left operand. |
| -= | Subtract right operand value from the left operand value and assigns the result to the left operand. |
| *= | Multiply left and right operand values and assigns the result to the left operand. |
| /= | Divide left operand value by right operand value and assign the result to the left operand. |
| %= | Get the modulus of left operand divide by right operand and assign resulted modulus to the left operand. |

Example: Assignment operators

```
let x = 5, y = 10, z = 15;

x = y; //x would be 10
```

```
x += 1; //x would be 6

x -= 1; //x would be 4

x *= 5; //x would be 25

x /= 5; //x would be 1

x %= 2; //x would be 1
```

### Ternary Operator

JavaScript provides a special operator called ternary operator :? that assigns a value to a variable based on some condition. This is the short form of the if else condition.

```
<condition> ? <value1> : <value2>;
```

The ternary operator starts with conditional expression followed by the ? operator. The second part (after ? and before :) will be executed if the condition turns out to be true. Suppose, the condition returns false, then the third part (after :) will be executed.

Example: Ternary operator

```
let a = 10, b = 5;

let c = a > b? a : b; // value of c would be 10
let d = a > b? b : a; // value of d would be 5
```


```
ue2>;
```

The ternary operator starts with conditional expression followed by the ? operator. The second part (after ? and before :) will be executed if the condition turns out to be true. Suppose, the condition returns false, then the third part (after :) will be executed.

Example: Ternary operator

```
let a = 10, b = 5;

let c = a > b? a : b; // value of c would be 10
let d = a > b? b : a; // value of d would be 5
```


**JavaScript Objects:**

Here you will learn objects, object literals, Object() constructor function, and access object in JavaScript.

You learned about primitive and structured data types in JavaScript. An object is a non-primitive, structured data type in JavaScript. Objects are same as variables in JavaScript, the only difference is that an object holds multiple values in terms of properties and methods.

In JavaScript, an object can be created in two ways: 1) using Object Literal/Initializer Syntax 2) using the Object() Constructor function with the new keyword. Objects created using any of these methods are the same.

The following example demonstrates creating objects using both ways.

Example: JavaScript Objects

```
var p1 = { name:"Steve" }; // object literal syntax

var p2 = new Object(); // Object() constructor function
p2.name = "Steve"; // property
```

Above, p1 and p2 are the names of objects. Objects can be declared same as variables using var or let keywords. The p1 object is created using the object literal syntax (a short form of creating objects) with a property named name. The p2 object is created by calling the Object() constructor function with the new keyword. The p2.name = "Steve"; attach a property name to p2 object with a string value "Steve".

Create Object using Object Literal Syntax

The object literal is a short form of creating an object. Define an object in the { } brackets with key:value pairs separated by a comma. The key would be the name of the property and the value will be a literal value or a function.

```
var <object-name> = { key1: value1, key2: value2,...};
```

The following example demonstrates objects created using object literal syntax.

Example: Object Literal Syntax

```
var emptyObject = {}; // object with no properties or methods

var person = { firstName: "John" }; // object with single property

// object with single method
var message = {
        showMessage: function (val) {
```

```
            alert(val);
        }
    };

// object with properties & method
var person = {
        firstName: "James",
        lastName: "Bond",
        age: 15,
        getFullName: function () {
            return this.firstName + ' ' + this.lastName
        }
    };
```

Note that the whole key-value pair must be declared. Declaring only a key without a value is invalid, as shown below.

### Example: Wrong Syntax

```
var person = { firstName };

var person = { getFullName: };
```

Create Objects using Objects() Constructor

Another way of creating objects is using the `Object()` constructor function using the `new` keyword. Properties and methods can be declared using the dot notation `.property-name` or using the square brackets `["property-name"]`, as shown below.

### Example: Create Object using Object() Constructor

```
var person = new Object();

// Attach properties and methods to person object
person.firstName = "James";
person["lastName"] = "Bond";
person.age = 25;
person.getFullName = function () {
    return this.firstName + ' ' + this.lastName;
};
```

An object can have variables as properties or can have computed properties, as shown below.

### Example: Variables as Object Properties

```
var firstName = "James";
var lastName = "Bond";

var person = { firstName, lastName }
```

Access JavaScript Object Properties & Methods

An object's properties can be accessed using the dot notation obj.property-name or the square brackets obj["property-name"]. However, method can be invoked only using the dot notation with the parenthesis, obj.method-name(), as shown below.

Example: Access JS Object

```
var person = {
            firstName: "James",
            lastName: "Bond",
            age: 25,
            getFullName: function () {
               return this.firstName + ' ' + this.lastName
            }
        };

person.firstName; // returns James
person.lastName; // returns Bond

person["firstName"];// returns James
person["lastName"];// returns Bond

person.getFullName(); // calling getFullName function
```

In the above example, the person.firstName access the firstName property of a person object. The person["firstName"] is another way of accessing a property. An object's methods can be called using () operator e.g. person.getFullName(). JavaScript engine will return the function definition if accessed method without the parenthesis.

Accessing undeclared properties of an object will return undefined. If you are not sure whether an object has a particular property or not, then use the hasOwnProperty() method before accessing them, as shown below.

Example: hasOwnProperty()

```
var person = new Object();

person.firstName; // returns undefined

if(person.hasOwnProperty("firstName")){
```

```
        person.firstName;
    }
```

The properties and methods will be available only to an object where they are declared.

### Example: Object Constructor

```
var p1 = new Object();
p1.firstName = "James";
p1.lastName  = "Bond";

var p2 = new Object();
p2.firstName; // undefined
p2.lastName; // undefined


p3 = p1; // assigns object
p3.firstName; // James
p3.lastName; // Bond
p3.firstName = "Sachin"; // assigns new value
p3.lastName = "Tendulkar"; // assigns new value
```

Enumerate Object's Properties

Use the for in loop to enumerate an object, as shown below.

### Example: Access Object Keys

```
var person = new Object();
person.firstName = "James";
person.lastName = "Bond";

for(var prop in person){
        alert(prop);  // access property name
        alert(person[prop]); // access property value
    };
```

Pass by Reference

Object in JavaScript passes by reference from one function to another.

### Example: JS Object Passes by Reference

```
function changeFirstName(per)
{
    per.firstName = "Steve";
```

```
}

var person = { firstName : "Bill" };

changeFirstName(person)

person.firstName; // returns Steve
```

Nested Objects

An object can be a property of another object. It is called a nested object.

Example: Nested JS Objects

```
var person = {
    firstName: "James",
    lastName: "Bond",
    age: 25,
    address: {
        id: 1,
        country:"UK"
    }
};

person.address.country; // returns "UK"
```