UNIT-3

JavaScript: Client side scripting, What is JavaScript, How to develop JavaScript, simple JavaScript, variables, functions, conditions, loops and repetition, Advance script, JavaScript and objects, JavaScript own objects, the DOM and web browser environments, forms and validations.

by : Pratishtha Gupta

Client-side Scripting

- Client-side scripting generates code that may be executed on the client end without needing server-side processing.
- These scripts are typically embedded into HTML text.
- Client-side scripting may be utilized to check the user's form for problems before submitting it and to change the content based on the user input.
- The web needs three components to function: client, database, and server.
- The client-side scripting may significantly reduce server demand.
- The HTML and CSS are delivered as plain text when a user uses a browser to request a webpage from the server, and the browser understands and renders the web content at the client end.

Client-side Scripting Languages

- There are various client-side scripting languages. Some main client-side scripting languages are:

HTML

- It is not a scripting language; it is a markup language. However, it serves as the basic language for client-side web development, also referred to as front-end.

CSS

- CSS provides a technique for creating graphic elements that help a web application's appearance look more appealing.

JavaScript

- It is a client-side scripting language designed for a specific purpose.

Features of Client-side Scripting

- There are various features of client-side scripting. Some main features of the client-side scripting are as follows:
- It is intended to execute code on which a web browser runs, and the results of the inputs are delivered to an accessible user.
- Client-side scripting enables greater involvement with clients via the browser and is used to validate programs and functionality based on the request.
- The client does not include any contact with the server in client-side scripting; the only interaction is receiving the requested data.

Javascript

JavaScript is the programming language of the Web.
JavaScript is easy to learn.

Why Study JavaScript?

JavaScript is one of the 3 languages all web developers must learn:

1. HTML to define the content of web pages.
2. CSS to specify the layout of web pages.
3. JavaScript to program the behavior of web pages

JavaScript Can Change HTML Content

One of many JavaScript HTML methods is getElementById().

JavaScript Can Change HTML Attribute Values

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

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

<h2>What Can JavaScript Do?</h2>

<p>JavaScript can change HTML attribute values.</p>

<p>In this case JavaScript changes the value of the src
(source) attribute of an image.</p>

<button
onclick="document.getElementById('myImage').src='pic_bulbon.gif
'">Turn on the light</button>

<img id="myImage" src="pic_bulboff.gif" style="width:100px">

<button
onclick="document.getElementById('myImage').src='pic_bulboff.gi
f'">Turn off the light</button>

</body>
</html>
```

JavaScript Can Change HTML Styles (CSS)

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

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

JavaScript Can Hide HTML Elements

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

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

JavaScript Can Show HTML Elements

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

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

The <script> Tag: In HTML, JavaScript code is inserted between <script> and </script> tags

```
<script>
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

JavaScript Functions and Events

A JavaScript function is a block of JavaScript code, that can be executed when "called" for. Eg: a function can be called when an event occurs, like when the user clicks a button.

JavaScript in <head> or <body>

- You can place any number of scripts in an HTML document.
- Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

JavaScript in <head>

- In this example, a JavaScript function is placed in the <head> section of an HTML page.
- The function is invoked (called) when a button is clicked:

JavaScript in <body>

- In this example, a JavaScript function is placed in the <body> section of an HTML page.
- The function is invoked (called) when a button is clicked:

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body>

<h2>Demo JavaScript in Head</h2>

<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

</body>
</html>
```

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

<h2>Demo JavaScript in Body</h2>

<p id="demo">A Paragraph</p>

<button type="button" onclick="myFunction()">Try it</button>

<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>

</body>
</html>
```

External JavaScript

- Scripts can also be placed in external files:
- External scripts are practical when the same code is used in many different web pages.
- JavaScript files have the file extension .js.
- To use an external script, put the name of the script file in the src (source) attribute of a <script> tag:
- You can place an external script reference in <head> or <body> as you like.
- The script will behave as if it was located exactly where the <script> tag is located.

External JavaScript Advantages

- Placing scripts in external files has some advantages:
- It separates HTML and code.

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

<h2>Demo External JavaScript</h2>

<p id="demo">A Paragraph.</p>

<button type="button" onclick="myFunction()">Try it</button>

<p>This example links to "myScript.js".</p>
<p>(myFunction is stored in "myScript.js")</p>

<script src="myScript.js"></script>

</body>
</html>
```

External file: myScript.js

```js
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
```

- It makes HTML and JavaScript easier to read and maintain
- Cached JavaScript files can speed up page loads

External References

- An external script can be referenced in 3 different ways:
- With a full URL (a full web address)

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

- With a file path (like /js/)

```
<script src="/js/myScript.js"></script>
```

- Without any path

```
<script src="myScript.js"></script>
```

- This example uses a full URL to link to myScript.js:

Javascript Variables

JavaScript Variables can be declared in 4 ways:

- Automatically
- Using var
- Using let
- Using const
- In this first example, x, y, and z are undeclared variables.
- They are automatically declared when first used.
- But, It is considered good programming practice to always declare variables before use.

```
<script>
x = 5;
y = 6;
z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script>
```

```
var x = 5;
var y = 6;
var z = x + y;
```

```
let x = 5;
let y = 6;
let z = x + y;
```

```
const x = 5;
const y = 6;
const z = x + y;
```

When to Use var, let, or const?

1. Always declare variables
2. use const if the value should not be changed
3. Always use const if the type should not be changed (Arrays and Objects)
4. Only use let if you can't use const
5. Only use var if you MUST support old browsers.

```
const price1 = 5;
const price2 = 6;
let total = price1 + price2;
```

Javascript

Javascript Identifiers

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter.
- Names can also begin with $ and _ (but we will not use it in this tutorial).
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.

Javascript Data Types

- JavaScript can handle many types of data, but for now, just think of numbers and strings.
- Strings are written inside double or single quotes. Numbers are written without quotes.
- If you put a number in quotes, it will be treated as a text string.

Javascript Assignment Operator

- In JavaScript, the equal sign (=) is an "assignment" operator, not an "equal to" operator.
- x = x + 5
- In JavaScript, however, it makes perfect sense: it assigns the value of x + 5 to x.
- (It calculates the value of x + 5 and puts the result into x. The value of x is incremented by 5.)

Javascript

Declaring a JavaScript Variable

- Creating a variable in JavaScript is called "declaring" a variable.
- You declare a JavaScript variable with the var or the let keyword:
  Eg: var carName; or let carName;
- After the declaration, the variable has no value (technically it is undefined).
- To assign a value to the variable, use the equal sign:
  Eg: carName = "Volvo";
- You can also assign a value to the variable when you declare it:
  Eg: let carName = "Volvo";
- You can declare many variables in one statement.
  Eg: let person = "John Doe", carName = "Volvo", price = 200;
- In computer programs, variables are often declared without a value. The value can be something that has to be calculated, or something that will be provided later, like user input.
- A variable declared without a value will have the value **undefined**.
- Eg: let carName; The variable carName will have the value undefined after the execution of this statement:

Re-Declaring JavaScript Variables

- If you re-declare a JavaScript variable declared with var, it will not lose its value.
- The variable carName will still have the value "Volvo" after the execution of these statements:

```
var carName = "Volvo";
var carName;
```

JavaScript Arithmetic

As with algebra, you can do arithmetic with JavaScript variables, using operators like = and +: Eg: let x = 5 + 2 + 3;

You can also add strings, but strings will be concatenated:

Example: let x = "John" + " " + "Doe";

```
{
  let x = 2;
}
// x can NOT be used here
```

Javascript let

- Variables declared with let have Block Scope
- Variables declared with let must be Declared before use
- Variables declared with let cannot be Redeclared in the same scope
- ES6 introduced the two new JavaScript keywords: let and const.
- These two keywords provided Block Scope in JavaScript
- Example
- Variables declared inside a { } block cannot be accessed from outside the block

Javascript

Global Scope

- Variables declared with the var always have Global Scope.
- Variables declared with the var keyword can NOT have block scope:

Example

- Variables declared with varinside a { } block can be accessed from outside the block:

```
{
  var x = 2;
}
// x CAN be used here
```

Cannot be Redeclared

Variables defined with let can not be redeclared.

You can not accidentally redeclare a variable declared with let.

Variables defined with var can be redeclared.

With `let` you **can not** do this:

```
let x = "John Doe";

let x = 0;
```

With `var` you **can** do this:

```
var x = "John Doe";

var x = 0;
```

Redeclaring Variables

- Redeclaring a variable using the var keyword can impose problems.

1. Redeclaring a variable inside a block will also redeclare the variable outside the block:

Javascript

2. Redeclaring a variable using the let keyword can solve this problem.

- Redeclaring a variable inside a block will not redeclare the variable outside the block:
- Redeclaring

3. Redeclaring a JavaScript variable with var is allowed anywhere in a program:

4. With let, redeclaring a variable in the same block is NOT allowed:

5. Redeclaring a variable with let, in another block, IS allowed:

```
var x = 10;
// Here x is 10

{
var x = 2;
// Here x is 2
}

// Here x is 2
```

1.

```
let x = 10;
// Here x is 10

{
let x = 2;
// Here x is 2
}

// Here x is 10
```

2.

3.

```
var x = 2;
// Now x is 2

var x = 3;
// Now x is 3
```

4.

```
var x = 2;   // Allowed
let x = 3;   // Not allowed

{
let x = 2;   // Allowed
let x = 3;   // Not allowed
}

{
let x = 2;   // Allowed
var x = 3;   // Not allowed
}
```

5.

```
let x = 2;   // Allowed

{
let x = 3;   // Allowed
}

{
let x = 4;    // Allowed
}
```

Javascript

JavaScript Const

- Variables defined with const cannot be Redeclared
- Variables defined with const cannot be Reassigned
- Variables defined with const have Block Scope

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

Cannot be Reassigned

- A variable defined with the const keyword cannot be reassigned:

Must be Assigned

- JavaScript const variables must be assigned a value when they are declared:

Constant Objects and Arrays

The keyword const is a little misleading.

It does not define a constant value. It defines a constant reference to a value.

Because of this you can NOT:

- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

But you CAN:

- Change the elements of constant array
- Change the properties of constant object

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

Javascript

Constant Objects

You can change the properties of a constant object:

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

cars = ["Toyota", "Volvo", "Audi"];    // ERROR

// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

// You can add a property:
car.owner = "Johnson";
```

```
const car = {type:"Fiat", model:"500", color:"white"};

car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

Block Scope

- Declaring a variable with const is similar to let when it comes to Block Scope.
- The x declared in the block, in this example, is not the same as the x declared outside the block:
- Redeclaring an existing var or let variable to const, in the same scope, is not allowed:

```
const x = 10;
// Here x is 10

{
const x = 2;
// Here x is 2
}

// Here x is 10
```

- Reassigning an existing const variable, in the same scope, is not allowed:
- Redeclaring a variable with const, in another scope, or in another block, is allowed

Hoisting

- Variables defined with var are hoisted to the top and can be initialized at any time.
- Meaning: You can use the variable before it is declared:

Javascript

```
var x = 2;       // Allowed
const x = 2;     // Not allowed

{
let x = 2;       // Allowed
const x = 2;     // Not allowed
}

{
const x = 2;     // Allowed
const x = 2;     // Not allowed
}
```

```
const x = 2;       // Allowed
x = 2;             // Not allowed
var x = 2;         // Not allowed
let x = 2;         // Not allowed
const x = 2;       // Not allowed

{
  const x = 2;     // Allowed
  x = 2;           // Not allowed
  var x = 2;       // Not allowed
  let x = 2;       // Not allowed
  const x = 2;     // Not allowed
}
```

```
const x = 2;        // Allowed

{
  const x = 3;    // Allowed
}

{
  const x = 4;    // Allowed
}
```

Example

This is OK:

```
carName = "Volvo";
var carName;
```

JavaScript Functions

- A JavaScript function is a block of code designed to perform a particular task.
- A JavaScript function is executed when "something" invokes it (calls it).

```
// Function to compute the product of p1 and p2
function myFunction(p1, p2) {
  return p1 * p2;
}
```

JavaScript Function Syntax

- A JavaScript function is defined with the function keyword, followed by a name and parentheses().
- Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).
- The parentheses may include parameter names separated by commas: (parameter1, parameter2, ...)
- The code to be executed, by the function, is placed inside curly brackets: {}

Javascript

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Function parameters are listed inside the parentheses () in the function definition.

Function arguments are the values received by the function when it is invoked.

Inside the function, the arguments (the parameters) behave as local variables.

Function Invocation

The code inside the function will execute when "something" invokes (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

Javascript

Function Return

- When JavaScript reaches a return statement, the function will stop executing.
- If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.
- Functions often compute a return value. The return value is "returned" back to the "caller":

```
// Function is called, the return value will end up in x
let x = myFunction(4, 3);

function myFunction(a, b) {
// Function returns the product of a and b
  return a * b;
}
```

The () Operator: The () operator invokes (calls) the function:

1. Accessing a function with incorrect parameters can return an incorrect answer:
2. Accessing a function without () returns the function and not the function result:

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}

let value = toCelsius(77);
```

1.

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}

let value = toCelsius();
```

Functions Used as Variable Values

Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

Example

Instead of using a variable to store the return value of a function, you can use the function directly, as a variable value:

2.

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}

let value = toCelsius;
```

```
let text = "The temperature is " + toCelsius(77) + " Celsius";
```

Javascript

Local Variables

- Variables declared within a JavaScript function, become LOCAL to the function.
- Local variables can only be accessed from within the function.
- Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.
- Local variables are created when a function starts, and deleted when the function is completed.

```
// code here can NOT use carName

function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Refer W3Schools for more details

Javascript

Conditional Statements: Very often when you write code, you want to perform different actions for different decisions. To do so, you can use conditional statements in your code
In JavaScript we have the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
- Use **else** to specify a block of code to be executed, if the same condition is false
- Use **else if** to specify a new condition to test, if the first condition is false
- Use **switch** to specify many alternative blocks of code to be executed

If Statement: Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax :

if (condition) {
// block of code to be executed if the condition is true
}

```
if (hour < 18) {
  greeting = "Good day";
}
```

The else Statement: Use the else statement to specify a block of code to be executed if the condition is false.

Syntax:

if (condition) {
// block of code to be executed if the condition is true
} else {
// block of code to be executed if the condition is false
}

```
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

The else if Statement: Use the else if statement to specify a new condition if the first condition is false.

Syntax:

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

Javascript

The JavaScript Switch Statement: Use the switch statement to select one of the code blocks to execute

Syntax:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

Example

- The getDay() method returns the weekday as a number between 0 and 6.
- (Sunday=0, Monday=1, Tuesday=2 ..)
- This example uses the weekday number to calculate the weekday name:
- The result of day will be Saturday

The break Keyword: When JavaScript reaches a break keyword, it breaks out of the switch block.

- This will stop the execution inside the switch block.
- It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.
- **Note: If you omit the break statement, the next case will be executed even if the evaluation does not match the case.**

The default Keyword: The default keyword specifies the code to run if there is no case match:

Example

- The getDay() method returns the weekday as a number between 0 and 6.
- If today is neither Saturday (6) nor Sunday (0), write a default message:

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

- The default case does not have to be the last case in a switch block:

Common Code Blocks

- Sometimes you will want different switch cases to use the same code.
- In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

Example:

Javascript

```
switch (new Date().getDay()) {
  case 4:
  case 5:
    text = "Soon it is Weekend";
    break;
  case 0:
  case 6:
    text = "It is Weekend";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

```
switch (new Date().getDay()) {
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

The result of text will be:

```
Today is Saturday
```

```
switch (new Date().getDay()) {
  default:
    text = "Looking forward to the Weekend";
    break;
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
}
```

Switching Details

- If multiple cases matches a case value, the first case is selected.
- If no matching cases are found, the program continues to the default label.
- If no default label is found, the program continues to the statement(s) after the switch.

Strict Comparison

- Switch cases use strict comparison (===).
- The values must be of the same type to match.
- A strict comparison can only be true if the operands are of the same type.
- In this example there will be no match for x:

```
let x = "0";
switch (x) {
  case 0:
    text = "Off";
    break;
  case 1:
    text = "On";
    break;
  default:
    text = "No value found";
}
```

JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

You can write:

```
for (let i = 0; i < cars.length; i++) {
  text += cars[i] + "<br>";
}
```

Instead of writing:

```
text += cars[0] + "<br>";
text += cars[1] + "<br>";
text += cars[2] + "<br>";
text += cars[3] + "<br>";
text += cars[4] + "<br>";
text += cars[5] + "<br>";
```

Javascript

Different Kinds of Loops

- JavaScript supports different kinds of loops:
- for - loops through a block of code a number of times
- for/in - loops through the properties of an object
- for/of - loops through the values of an iterable object
- while - loops through a block of code while a specified condition is true
- do/while - also loops through a block of code while a specified condition is true

The For Loop

The for statement creates a loop with 3 optional expressions:

Syntax:

```
for (expression 1; expression 2; expression 3) {
  // code block to be executed
}
```

Example

```
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

Expression 1 is executed (one time) before the execution of the code block.
Expression 2 defines the condition for executing the code block.
Expression 3 is executed (every time) after the code block has been executed.
From the example above, you can read:
Expression 1 sets a variable before the loop starts (let i = 0).
Expression 2 defines the condition for the loop to run (i must be less than 5).
Expression 3 increases a value (i++) each time the code block in the loop has been executed.

Expression 1: Normally you will use expression 1 to initialize the variable used in the loop (let i = 0).

- This is not always the case. JavaScript doesn't care. Expression 1 is optional.
- You can initiate many values in expression 1 (separated by comma):

Example: for (let i = 0, len = cars.length, text = ""; i < len; i++)

- And you can omit expression 1 (like when your values are set before the loop starts):

```
let i = 2;
let len = cars.length;
let text = "";
for (; i < len; i++) {
  text += cars[i] + "<br>";
}
```

Example:

Expression 2: Often expression 2 is used to evaluate the condition of the initial variable.

- This is not always the case. JavaScript doesn't care. Expression 2 is also optional.
- If expression 2 returns true, the loop will start over again. If it returns false, the loop will end.
- If you omit expression 2, you must provide a break inside the loop. Otherwise the loop will never end. This will crash your browser.

Expression 3: Often expression 3 increments the value of the initial variable.

- This is not always the case. JavaScript doesn't care. Expression 3 is optional.
- Expression 3 can do anything like negative increment (i--), positive increment (i = i + 15), or anything else.
- Expression 3 can also be omitted (like when you increment your values inside the loop):

```
let i = 0;
let len = cars.length;
let text = "";
for (; i < len; ) {
  text += cars[i] + "<br>";
  i++;
}
```

Loop Scope:

- i. Using var in a loop:
- ii. Using let in a loop:
- In the first example, using var, the variable declared in the loop redeclares the variable outside the loop.
- In the second example, using let, the variable declared in the loop does not redeclare the variable outside the loop.
- When let is used to declare the i variable in a loop, the i variable will only be visible within the loop.

```
var i = 5;

for (var i = 0; i < 10; i++) {
  // some code
}

// Here i is 10
```

```
let i = 5;

for (let i = 0; i < 10; i++) {
  // some code
}

// Here i is 5
```

The For In Loop: The JavaScript for in statement loops through the properties of an Object:

Syntax:

for (key in object) {
* // code block to be executed*
}

```
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person) {
  text += person[x];
}
```

Example Explained

- The for in loop iterates over a person object
- Each iteration returns a key (x)
- The key is used to access the value of the key
- The value of the key is person[x]

For In Over Arrays: The JavaScript for in statement can also loop over the properties of an Array:

Syntax:

for (variable in array) {
code
}

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

Array.forEach()

The forEach() method calls a function (a callback function) once for each array element.

The For Of Loop: The JavaScript for of statement loops through the values of an iterable object.
It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more:

Syntax:

for (variable of iterable) {
// code block to be executed
}

variable - For every iteration the value of the next property is assigned to the variable. Variable can be declared with const, let, or var.
iterable - An object that has iterable properties.

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);

function myFunction(value) {
  txt += value;
}
```

Looping over an Array

```
const cars = ["BMW", "Volvo", "Mini"];

let text = "";
for (let x of cars) {
  text += x;
}
```

Looping over a String

```
let language = "JavaScript";

let text = "";
for (let x of language) {
text += x;
}
```

The While Loop: The while loop loops through a block of code as long as a specified condition is true.

Syntax:

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

Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

Note: if you don't increment the variable used in the condition, the loop will never end. This will crash your browser.

The Do While Loop: The do while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax:

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

Example

The example below uses a do while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The break statement "jumps out" of a loop.
The continue statement "jumps over" one iteration in the loop.

The Break Statement

Earlier we used break statement to break out of switch case. It can also be used to jump out of a loop:

In the example, the break statement ends the loop ("breaks" the loop) when the loop counter (i) is 3.

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
```

The Continue Statement

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 3:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { continue; }
  text += "The number is " + i + "<br>";
}
```

JavaScript Labels

To label JavaScript statements you precede the statements with a label name and a colon:

Syntax:

break labelname;

continue labelname;

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];
list: {
  text += cars[0] + "<br>";
  text += cars[1] + "<br>";
  break list;
  text += cars[2] + "<br>";
  text += cars[3] + "<br>";
}
```

The continue statement (with or without a label reference) can only be used to skip one loop iteration.

The break statement, without a label reference, can only be used to jump out of a loop or a switch.

With a label reference, the break statement can be used to jump out of any code block:

OBJECT
- Properties (data)
- Methods (functions)

Javascript Objects

- Real Life Objects
- In real life, objects are things like: houses, cars, people, animals, or any other subjects.
- Here is a car object example:
- Object Properties
- A real life car has properties like weight and color:
- car.name = Fiat, car.model = 500, car.weight = 850kg, car.color = white.
- Car objects have the same properties, but the values differ from car to car.

Car Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

Object Methods

- A real life car has methods like start and stop:
- car.start(), car.drive(), car.brake(), car.stop().
- Car objects have the same methods, but the methods are performed at different times.

JavaScript Objects

- Objects are variables too. But objects can contain many values.
- This code assigns many values (Fiat, 500, white) to an object named car:
- Example
- const car = {type:"Fiat", model:"500", color:"white"};

JavaScript Object Definition

- **How to Define a JavaScript Object**
- Using an Object Literal
- Using the new Keyword
- Using an Object Constructor

1. **JavaScript Object Literal**

- An object literal is a list of name:value pairs inside curly braces {}.
- {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}

Javascript

Creating a JavaScript Object

- These examples create a JavaScript object with 4 properties:
- Examples
- // Create an Object
- const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

2. **Using the new Keyword**

This example create a new JavaScript object using new Object(), and then adds 4 properties:

const person = new Object();

```
// Create an Object
const person = new Object();

// Add Properties
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

Accessing Object Properties

- You can access object properties in two ways:
- objectName.propertyName
- objectName["propertyName"]

JavaScript Object Methods

- Methods are actions that can be performed on objects.
- Methods are function definitions stored as property values.

- In the example above, this refers to the person object:
- this.firstName means the firstName property of person.
- this.lastName means the lastName property of person.

```
const person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```


JavaScript Objects are Mutable

- Objects are mutable: They are addressed by reference, not by value.
- If person is an object, the following statement will not create a copy of person:
- Eg: const x = person;
- The object x is not a copy of person. The object x is person.
- The object x and the object person share the same memory address.
- Any changes to x will also change person:

Javascript

An Object is an Unordered Collection of Properties

Properties are the most important part of JavaScript objects.

Properties can be changed, added, deleted, and some are read only.

Accessing JavaScript Properties

The syntax for accessing the property of an object is:

```
let age = person.age; // objectName.property
let age = person["age"]; //objectName["property"]
let age = person[x]; //objectName[expression]
```

Adding New Properties: You can add new properties to an existing object

Eg: person.nationality = "English";

Deleting Properties: The delete keyword deletes a property from an object:

Eg:

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

Javascript

JavaScript Object Methods

- Object methods are actions that can be performed on objects.
- A method is a function definition stored as a property value.

Accessing Object Methods

- You access an object method with the following syntax:
- objectName.methodName()
- If you invoke the fullName property with (), it will execute as a function:
- Eg: name = person.fullName();
- If you access the fullName property without (), it will return the function definition:
- Eg: name = person.fullName;

Adding a Method to an Object

- Adding a new method to an object is easy:
- Eg: person.name = function () {
- return this.firstName + " " + this.lastName;
- };

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

How to Display JavaScript Objects?

Displaying a JavaScript object will output [object Object].

Eg: const person = {
 name: "John",
 age: 30,
 city: "New York"
};

document.getElementById("demo").innerHTML = person;

Displaying Object Properties: The properties of an object can be displayed as a string:

Eg: const person = {
 name: "John",
 age: 30,
 city: "New York"
};

// Display Properties

document.getElementById("demo").innerHTML =
person.name + "," + person.age + "," + person.city;

Javascript

Displaying Properties in a Loop

The properties of an object can be collected in a loop:

Eg:

```
const person = {
 name: "John",
 age: 30,
 city: "New York"
};

// Build a Text
let text = "";
for (let x in person) {
 text += person[x] + " ";
};

// Display the Text
document.getElementById("demo").innerHTML = text;
```

Javascript

Using Object.values()

Object.values() creates an array from the property values:

```
const myArray = Object.values(person);
```

```
// Create an Object
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

// Create an Array
const myArray = Object.values(person);

// Display the Array
document.getElementById("demo").innerHTML = myArray;
```

Using Object.entries()

Object.entries() makes it simple to use objects in loops:

```
let text="";
for (let [fruits, value] of Object.entries(fruits)) {
   text +=fruit + ":" + value + "<br>";
}
```

```
const fruits = {Bananas:300, Oranges:200, Apples:500};

let text = "";
for (let [fruit, value] of Object.entries(fruits)) {
  text += fruit + ": " + value + "<br>";
}
```

Javascript

Using JSON.stringify()

JavaScript objects can be converted to a string with JSON method JSON.stringify().
JSON.stringify() is included in JavaScript and supported in all major browsers.

```
let myString = JSON.stringify(person);
```

```
// Create an Object
const person = {
  name: "John",
  age: 30,
  city: "New York"
};

// Stringify Object
let myString = JSON.stringify(person);

// Display String
document.getElementById("demo").innerHTML = myString;
```

Object Constructor Functions

Sometimes we need to create many objects of the same type.
To create an object type we use an object constructor function.
It is considered good practice to name constructor functions with an upper-case first letter.

Now we can use new Person() to create many new Person objects:

Eg: const myFather = new Person("John", "Doe", 50, "blue");

const myMother = new Person("Sally", "Rally", 48, "green");

Property Default Values

A value given to a property will be a default value for all objects created by the constructor: Eg: this.nationality = "English";

Adding a Property to an Object

Adding a property to a created object is easy:

Eg: myFather.nationality = "English";

Adding a Property to a Constructor

You can NOT add a new property to an object constructor:

Eg: Person.nationality = "English";

To add a new property, you must add it to the constructor function prototype:

Eg: Person.prototype.nationality = "English";

Constructor Function Methods

A constructor function can also have methods:

Object Type Person

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}
```

Adding a Method to an Object

Adding a method to a created object is easy:

Eg: myMother.changeName = function (name) {
this.lastName = name;
}

Adding a Method to a Constructor

You cannot add a new method to an object constructor function.

This code will produce a TypeError:

```
Person.changeName = function (name) {
  this.lastName = name;
}

myMother.changeName("Doe");
```

```
TypeError: myMother.changeName is not a function
```

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
  this.fullName = function() {
    return this.firstName + " " + this.lastName;
  };
}
```

Adding a new method must be done to the constructor function prototype:

Eg: Person.prototype.changeName = function (name) {
this.lastName = name; }
myMother.changeName("Doe");

Javascript

Web Browser Environment

- JavaScript can interact with the browser environment in various ways:
- Window Object
- The window object represents the browser's window.

- **Alert, Confirm, and Prompt:**

```javascript
window.alert("Hello, World!");
let result = window.confirm("Are you sure?");
let name = window.prompt("Enter your name:");
```

- **Timers:**

```javascript
setTimeout(function() {
  console.log("This runs after 2 seconds");
}, 2000);

let interval = setInterval(function() {
  console.log("This runs every 2 seconds");
}, 2000);

clearInterval(interval); // Stops the interval
```

Purpose: Runs the provided function once after a delay.

Delay: 2000 milliseconds (i.e., 2 seconds).

Output: After 2 seconds, the message "This runs after 2 seconds" is printed to the console.

Purpose: Runs the provided function repeatedly at every interval (in this case, every 2 seconds).

Output: Every 2 seconds, it prints "This runs every 2 seconds" to the console.

Purpose: Stops the interval set by setInterval().

Effect: The interval won't actually run even once, because clearInterval(interval) is called immediately after setting it, cancelling it before it has a chance to execute.

Navigator Object

- The navigator object contains information about the browser.
- **Checking Browser Information:**

```
let appName = navigator.appName;
let appVersion = navigator.appVersion;
let userAgent = navigator.userAgent;
```

- **Location Object**
- The location object contains information about the current URL.
- **Getting URL Information:**

```javascript
let currentURL = location.href;
let hostName = location.hostname;
let pathName = location.pathname;
```

- **Redirecting:**

```javascript
location.href = "https://www.example.com";
```

JavaScript Form Validation

Javascript

- 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;
  }
}
```

```
<!DOCTYPE html>
<html>
<head>
<script>
function validateForm() {
  let x = document.forms["myForm"]
["fname"].value;
  if (x == "") {
    alert("Name must be filled out");
    return false;
  }
}
</script>
</head>
<body>
<h2>JavaScript Validation</h2>
<form name="myForm" action="/action_page.php"
onsubmit="return validateForm()"
method="post">
  Name: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>

</body>
</html>
```

```
<script>
function myFunction() {
  // Get the value of the input field with id="numb"
  let x = document.getElementById("numb").value;
  // If x is Not a Number or less than one or greater than 10
  let text;
  if (isNaN(x) || x < 1 || x > 10) {
    text = "Input not valid";
  } else {
    text = "Input OK";
  }
  document.getElementById("demo").innerHTML = text;
}
</script>
```

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:

```
<h2>JavaScript Validation</h2>

<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