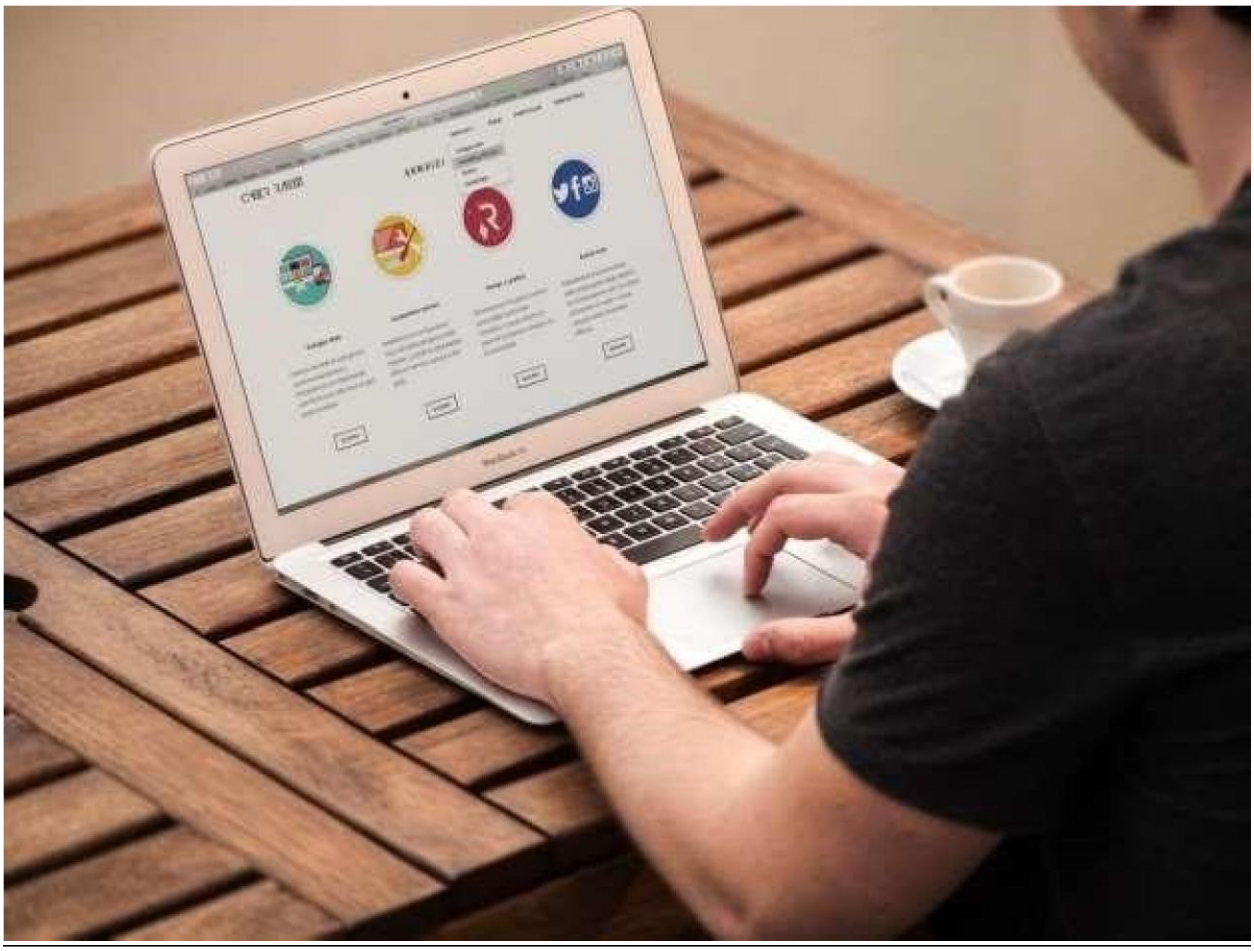


JavaScript

Developed by Alabian Solutions Ltd

Table of Contents

CHAPTER 1

Introduction to JavaScript

JavaScript is the world's most popular programming language. JavaScript is the programming language of the Web.

You can work as a frontend developer, backend developer and also as a full-stack developer with JavaScript.

For a very long JavaScript was only used in the browser to build interactive web pages but those days are gone due to heavy investment from big companies like Google. Nowadays you can build full grown web/mobile applications, real-time networking applications, games, etc.

JavaScript was originally designed to run on browsers. That means every browser has a JavaScript engine used to execute JavaScript code. For example, the JavaScript engine for Firefox is Spider-Monkey while that of Chrome is the V8 engine.

In 2009, the JavaScript engine in Chrome was embedded inside a C++ program, and was called Node. So, Node is a C++ program that includes Google Chrome V8 engine. We can also say Node is a JavaScript runtime built on Chrome's V8 JavaScript engine. Now with this we can run JavaScript program outside of a browser. We can pass JavaScript codes into Node, that means we can build backend applications like web and mobile apps.

In a nutshell, JavaScript code can be run inside of a browser or in a terminal (Node). Browser and Node provides a run-time environment for our JavaScript code.

Brief History of JavaScript

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997. JavaScript is a programming language. JavaScript files have the file extension .js. ECMA are responsible for defining standards. The first version of ECMAScript was released in 1997 which is called V1 and the next in 2015, 2016 which was called ES6.

Note: JavaScript and Java are completely different languages, both in concept and design.

Requirements to Learning JavaScript

- A basic knowledge of HTML and CSS
- A text editor
- A browser
- Logical thinking and Tenacity

Why Learn JavaScript?

1. It is the most commonly used programming language
2. It is easy to learn
3. It creates functional and dynamic web pages/app
4. It is used for websites, mobile apps, frontend/backend
5. Jobs in high demands
6. It has frameworks and libraries, e.g., React, Vue, Angular, Node

CHAPTER 2

Setting Up Your JavaScript Environment

To set up your JavaScript environment, you will definitely need the following:

- Text editor – E.g., Visual Studio Code
- Browser – E.g., Chrome
- Node – (Not mandatory)

Running JavaScript on Browser

You can run your JavaScript code on browser by using the console. First right click your Chrome's browser and click on "Inspect" in the dialog box, automatically your console window will open. You can then run your JavaScript code on the console.

```
console.log("Hello World!");
```

Running JavaScript on Browser

- Open a terminal in which your folder is the main directory
- Type "node index.js" and press "Enter"

Node is a separate topic entirely, so we will not be focusing on node in this course.

Ways to Embed JavaScript code in HTML

JavaScript embed code

Just like in CSS, we can embed JavaScript code using the following methods:

1. Inline JavaScript
2. Internal JavaScript
3. External JavaScript

Inline JavaScript

They are JavaScript code embedded in a HTML document**.** It can save the web browser round trips to the server. they can be difficult to maintain over time if they involve multiple pages. We talk more about it when will reach DOM.

```
<!DOCTYPE html>
<html lang="en">

<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
</head>

<body>
   <button onclick="alert('Hello world')">Click Me</button>
</body>

</html>
```

Internal JavaScript

JavaScript can be added directly to the HTML file by writing the code inside the <script> tag. We can place the <script> tag inside the <body> tag according to the need. Its separate HTML and JavaScript code. It makes HTML and JavaScript easier to read and maintain. It is rarely used for multiple pages.

```html
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
 </head>
 <body>
  <h1>Welcome to JavaScript</h1>

  <script>
   console.log("hello world!");
  </script>
 </body>
</html>
```

Note: When you place your JavaScript at the bottom of your HTML body, it gives the HTML time to load before any of the JavaScript loads, which can prevent errors, and speed up website response time.

External JavaScript

JavaScript can also be placed in external files. To use an external script, put the name of the script file in the src (source) attribute of a <script> tag.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <h1>Hello World!</h1>
    <script src="hello.js"></script>
</body>

</html>
```

JavaScript Outputs

JavaScript can "display" data in different ways:

- Writing into the browser console, using console.log()

```
console.log("hello world!");
```

CHAPTER 3

JavaScript Statements

A JavaScript program is made up of a series of statements. These statements contain JavaScript language construct, keyword and command. Each statement ends with a new line or semicolon. But a popular convention adopted by JavaScript programmer is to write one statement per line and end the line with a semi colon.

```
let name = "Janet";
```

JavaScript Comments

Comments are human-readable texts ignored by the computer. There are two types of comment in JavaScript:

1. Single Line comment – This is used for a single line
2. Multiple Line comment – This is used for multiple lines

```
SINGLE LINE COMMENT
// This is a single comment
MULTIPLE LINE COMMENTS
/*
This is a multiple line comment on line one
This is a multiple line comment on line two
This is a multiple line comment on line three
*/
```

Variables

Variable is a reference to a memory location in the computer memory (RAM) usually for storage of data. We perform three (3) actions with variable:

1. Declare the variable
2. Put data inside the variable
3. Get the data inside the variable

JavaScript uses "**var**", **"let"** and **"const"** keywords to declare variables. There are issues associated with variables declared with the var keywords. In this tutorial we will not use the var keyword.

The keyword "**const**" is used when the variable will not be reassigned to another value, whereas "let" is used if the variable might be reassigned later in the program.

Differences/Similarity between const and let variable keywords

Const keyword	Let keyword
const cannot be re-declared	let cannot be redeclared
const cannot be re-assigned a value	let can be re-assigned a value

```
let name = "Seyi";
name = "Janet";
console.log(name);
```

If you noticed the "name" variable above was reassigned another value (from Seyi to Janet)

```
const PI = 3.14;
PI = 9.8;
console.log(PI);

/* This will output an error message
   saying "Assignment to constant variable."
   because in a const keyword you
   cannot reassigned a variable
   to another value
*/
```

Variables are usually given a name so that it is easy to differentiate them. There are some rules that must be followed when giving variable name.

Variable naming rules

1. They can only contain the following character; letters, numbers, underscores or dollar signs but the first character cannot be a number

2. They are case sensitive so ‘numberOne’ is different from ‘NumberOne’.

3. They cannot be JavaScript reserved words.

4. Choose clarity and meaning

5. Since they cannot contain space variable name that compose of more than a word must be joined together. So ‘number one’ will be written as ‘numberone’. But for ease of readability ‘numberone’ is usually written as ‘numberOne’ or ‘number_one’. The camel case (ie numberOne) convention is preferred by many JavaScript programmers.

6. Pick a naming convention and stick with it.

```
let 2numberOne; // invalid variable name because it started with number
let numberOne!; // invalid variable name because it contains a non permitted character
let x; // valid variable name but not descriptive
let numberOne; // variable declaration
let numberTwo; // variable declaration
numberOne = 5; // assigning data to variable
numberTwo = 10 // assigning data to variable
```

CHAPTER 4

Data Types

JavaScript has five primitive data types, which are string, number, boolean, null and undefined. And one non-primitive data types which is object. There is also another data type, which is array but is a special form of object data type. Also, string, number and boolean data types can also be expressed as object data type, more on this later in the manual.

There are two categories of data types:

1. Primitive data type

2. Non primitive data type

Primitive data type

Primitive data types consist of string, number, Boolean, null, and undefined.

1. String data type
2. Number data type
3. Boolean data type
4. Null
5. Undefined

- **String data type:** A string is a collection of letters and symbols enclosed within quote mark. The quote mark can either be single or double. If you want to use double quote marks inside the string, you need to use single quote marks to enclose the string. And if you want to use an apostrophe in your string, you need to employ double quote marks to enclose the string.
- **Number Data Type:**
  Number can be integers (whole numbers, such as 3) or floating point decimals (often referred to as just "decimals" or "floats", such as 3.14159).
  The only character permitted as number apart from digitals (0-9) are dot(.) and exponential(e or E). NOTE comma(,) is not a valid character in number in JavaScript.

- **Boolean Data Type:**
  There are only two boolean values which are ‘true’ and ‘false’. Boolean values are fundamental in the logical statements that make up a computer program. Every value in JavaScript has a Boolean value.

- **Undefined and Null Data Type:**
  Undefined is the value given to variables that have been declared but not yet assigned a value. It can also occur if an object’s property doesn’t exist or a function has a missing parameter.
  While null represents a non-exiting data, more like a call to a variable that has not be declared before. So that variable does not represent any reference to any location in the computer memory.

Operation of Primitive Data type

String data type

```
  let name = "Hello World";
      console.log(name);//it returns "Hello World"
      console.log(typeof name);//This returns string
```

```
let name = 'John Doe'
let bookTitle = "Things fall apart"
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

```

// Single quote inside double quotes:
let answer1 = "It's alright";

// Single quotes inside double quotes:
let answer2 = "He is called 'Johnny'";

// Double quotes inside single quotes:
```

```
let answer3 = 'He is called "Johnny"';
```

```
let academy = 'Alabian Solution\'s ex-students are the best'; //
You place a backslash before the apostrophe to escape it
```

String concatenation

In JavaScript, we can assign strings to a variable and use concatenation to combine the variable to another string. To concatenate a string, you **add a plus sign+ between the strings or string variables you want to connect**

Example:

```
let partSentenceOne = "This is a ";
let partSentenceTwo = "complete sentence";
let output = partSentenceOne + partSentenceTwo
```

```
let num1 = "3";
let num2 = "4";
const result = num1 + num2;
console.log(result); // This will return 34 due to concatenation
```

Note: You can only concatenate a string.

Template Literals

Template literals (template strings) allow you to use strings or embedded expressions in the form of a string. They are enclosed in backticks (``).

You can use Template Literals (which was introduced in ES6).

Example:

```
const name = "Jack";
console.log(`Hello ${name}!`); // Hello Jack!
```

Template Literals can also be used as expression interpolation.

Before JavaScript ES6, you would use the + operator to concatenate variables and expressions in a string.

Example:

```
let name = "Mark";
let companyName = "Alabian Solution Limited";

// Normal concatenation
console.log(name + " works at " + companyName);

// Template Literals
console.log(`${name} works at ${companyName}`);
```

Noted: JavaScript is loosely typed, meaning you don't have to tell it the data type of a certain variable being declared unlike Java or C++. Because of this it will try helping you out in what we called implicit type conversion.

Number data type

Example:

```
let number = 23;
console.log(typeof number);
234; // integer
45.6; // float
1e6; // means 1 multiplied by 10 to the power 6 (a million)
NaN; // usually called Not a Number
```

Operators

Operators are used to assign values, compare values, perform arithmetic operations and more

Types of operators

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators

Arithmetic Operators are used to perform arithmetic on numbers:

Arithmetic Operators	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus
++	Increment
--	Decrement

Examples:

+ Operator: The **addition** operator (+) adds numbers:

```
let x = 5;
let y = 2;
let z = x + y;
```

- Operator: The **subtraction** operator (-) subtracts numbers.

```
let x = 5;
let y = 2;
let z = x - y;
```

***Operator:** The multiplication operator (*) multiplies numbers.

```
let x = 5;
let y = 2;
let z = x * y;
```

/ Operator: The division operator (/) divides numbers

```
let x = 5;
let y = 2;
let z = x / y;
```

% Operator: The modulus operator (%) returns the division remainder.

```
let x = 5;
let y = 2;
let z = x % y;
```

++ Operator: The increment operator (++) increments numbers.

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

-- Operator: The decrement operator (--) decrements numbers.

```
let x = 5;
x--;
let z = x;
```

**** Operator:** The exponentiation operator (**) raises the first operand to the power of the second operand.

```
let x = 5;
let z = x ** 2;
```

Assignment operators assign values to JavaScript variables.

```
let x = 10;
x += 5;
```

Boolean data type

There are only two boolean values which are ‘true’ and ‘false’. Boolean always return either true or false

Example:

```
let bool = true;
console.log(typeof bool);// boolean
```

Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.

Comparison Operators	Description
Equal to (==)	Returns true if the operands are equal.
Not Equal to (! =)	Returns true if the operands are not equal.
Strict Equal to (= = =)	Returns true if the operands are equal and of the same type.
Strict not Equal to (! = =)	Returns true if the operands are equal of the same type but not equal, or are of different type.
Greater than (>)	Returns true if the left operand is greater than the right operand.
Greater than or equal to (>=)	Returns true if the left operand is greater than or equal to the right operand.
Less than (<)	Returns true if the left operand is less than the right operand.
Less than or equal to (<=)	Returns true if the left operand is less than or equal to the right operand.

Examples:

== Operator: Equal to

```
let x = 5;
x == 5; // This will return true

x == '5' // This will return true

x == '8' // This will return false
```

!= Operator: Not equal

```
let x = 5;
x != 8; // This will return true
```

=== Operator: Strict Equal

```
let x = 5;
x === 5; // This will return true

x === '5' // This will return false
```

!== Operator: Strict not Equal to

```
let x = 5;

x !== 8; // This will return true

x !== '5'; // This will return true

x !== 5; // This will return false
```

- **Operator:** Greater than

```
let x = 5;

x > 8; // This will return false
```

< Operator: Less than

```
let x = 5;

x < 8; // This will return true
```

>= Operator: Greater than or equal to:

```
let x = 5;

x >= 8; // This will return false
```

<= Operator: Less than or equal to

```
let x = 5;

x <= 8; // This will return false
```

Logical Operators

Logical operators are used to test for true or false.

Logical Operator	Description
&&	Logical and
||	Logical or
!	Logical not

Examples:

&& Logical AND Operator:

```
let x = 6;
let y = 3;
let z = x < 10 && y > 1 //This will return true
```

|| Logical OR Operator:

```
x = 6;
let y = 3;
let z = x == 5 || y == 5 // This will return false
```

! Logical NOT Operator

```
let x = 6;
let y = 3;
let z = !(x==y) // This will return true
```

Undefined and Null Data Type.

Undefined is the value given to variables that have been declared but not yet assigned a value. It can also occur if an object's property doesn't exist or a function has a missing parameter.

While null represents a non-exiting data, more like a call to a variable that has not be declared before. So that variable does not represent any reference to any location in the computer memory.

Example:

```
let x
console.log(x);
```

CHAPTER 5

Making Decision by Conditional statements

The conditional statements included in the JavaScript code assist with decision making, based on certain conditions. The condition specified in the conditional statement can either be true or false. The conditional statements execute the associated piece of code only if the condition is true. We have four types of conditional statements in JavaScript:

1. An **if statement** executes a specified code segment if the given condition is "true."

2. An **else statement** executes the code associated with the else if the given if condition is "false."

3. An **else if statement** specifies a new condition for testing when the first if condition is "false."

4. A **switch-case statement** specifies multiple alternative code blocks for execution and executes these cases based on certain conditions.

IF statement

IF statement is used specify a block of code to be executed, if a specified condition is true.

Syntax of the if statement.

```
if (condition) {
  // code to run if condition is true
}
```

Example:

```
const age = 12;
if (age < 18) {
  console.log("Sorry, you are not old enough to play");
}
```

IF and ELSE statement

Use else to specify a block of code to be executed, if the same condition is false.

Example:

```
const price = 1500;
if (price < 2000) {
  console.log("Sorry, you do not have enough cash to purchase
this product");
} else {
  console.log("else top up your account");
}
```

IF…ELSE IF statement

Use else if to specify a new condition to test, if the first condition is false Code:

Example:

```
const price = 1500;
if (price == 2000) {
  console.log("You just purchased a crate of egg");
} else if (price == 1500) {
  console.log("You just purchased a bottle of cashew nut");
} else {
  console.log("You have no cash");
}
```

Ternary Operator

This is an alternative way or shorthand in writing the if …else statement.

```
const age = 19;
age > 18 ? alert("Adult") : alert("Child");
```

SWITCH Statement

Use the switch statement to select one of many code blocks to be executed.

Syntax:

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

```
// program using switch statement
let a = 2;
switch (a) {
  case 1:
    a = "one";
    break;
  case 2:
    a = "two";
    break;
  default:
    a = "not found";
    break;
}
console.log("The value is " + a);
```

Combining conditional statements with JavaScript operators

Examples:

```
const user = "Busayo";
const age = "18";
if (user == "Busayo" && age >= 18) {
  alert("You are an adult");
} else {
  alert("You are not an adult");
}
```

```
const user = "Busayo";
const age = "18";
if (user == "Busayo" || age >= 18) {
  alert("You are an adult");
} else {
  alert("You are not an adult");
}
```

CHAPTER 6

LOOPS

Loops will repeat a piece of code repeatedly according to certain conditions.

A loop should have when to start, stop and how to get to the next item. Loops help us to access a value in an array or object.

Why do will need Loop?

Loops are used in JavaScript to perform repeated tasks based on a condition. Conditions typically return true or false. A loop will continue running until the defined condition returns false.

In this case, you will use a loop. This is the important of a loop.

Kinds of JavaScript Loops

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

WHILE Loops

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:

```
// while loop
let i = 1;
let n = 5;

while (i <= n) {
  console.log(i);
  i += 1;
}
```

Output:

```
1
2
3
4
5
```

Note: If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

FOR Loops

For loop loops through a block of code a number of times.

The syntax:

```
for (Start; StopCondition; move) {
  // code block to be executed
}
```

Start (Initialization): - This expression runs before the execution of the first loop, and is usually used to create a counter.

Stop Condition: - This expression is checked each time before the loop runs. If it evaluates to true, the statement or code in the loop is executed. If it evaluates to false, the loop stops. And if this expression is omitted, it automatically evaluates to true.

Move: - This expression is executed after each iteration of the loop. This is usually used to increment a counter, but can be used to decrement a counter instead.

Example:

```
for (let i = 0; i <=10; i++) {
  console.log(i);
}
```

Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Start: - Sets a variable before the loop starts (let i = 0).

Stop Condition: - Defines the condition for the loop to run (i must be less than or equal to 10).

Move: - Increases a value (i++) each time the code block in the loop has been executed.

```
for (let i = 0; i < 5; i++) {
  console.log("The number is " + i);
}
```

Output:

```
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
```

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:

```
// do while loop
let i = 0;
do {
  i++;
  console.log("This is number " + i);
} while (i <= 5);
```

Output:

```
This is number 1
This is number 2
This is number 3
This is number 4
This is number 5
This is number 6
```

Break statement

The break statement breaks out of a switch or a loop. In a switch, it breaks out of the switch block. This stops the execution of more code inside the switch. In in a loop, it breaks out of the loop and continues executing the code after the loop (if any).

Example:

```
for (let i = 0; i < 5; i++) {
  if (i == 3) {
    console.log("i is equal to" + " " + i);
    break;
  }
  console.log(i);
}
```

Output:

```
0
1
2
i is equal to 3
```

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.

The difference between continue and the break statement, is instead of "jumping out" of a loop, the continue statement "jumps over" one iteration in the loop.

Example:

```
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    console.log("i is equal to" + " " + i);
    continue;
  }
  console.log(i);
}
```

Output:

```
0
1
2
i is equal to 3
4
```

Example:

```
for (let i = 1; i < 10; i++) {
  if (i % 2 == 0) {
    console.log(i + " is even number ");
    continue;
  }
  console.log(i + " is odd number ");
}
```

Output:

```
1 is odd number
2 is even number
3 is odd number
4 is even number
5 is odd number
6 is even number
7 is odd number
8 is even number
9 is odd number
```

CHAPTER 7

Function

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

A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses **( )**.

Function names can contain letters, digits, underscores, and dollar signs.

Why Functions?

You can reuse code: Define the code once, and use it many times. You can use the same code many times with different arguments, to produce different results.

Function declaration and expression

Function declaration

A function **declaration** defines a function that will be executed when it is invoked.

Examples:

```
function people() {
  console.log("Hello world!");
}
// Invoke the function
people();

//Output: Hello world!
```

Function expression

A function **expression** is similar to a function declaration, with the difference that it can be stored in a variable. As soon as the function is defined with an expression, it is invoked.

Examples:

```
const people = function() {
  console.log("Hello world!");
}
// Invoke the function
people();

//Output: Hello world!
```

Parameters and Arguments

Parameters and arguments are often used interchangeably to represent values that are provided for the function to use. Let's break this down, parameters are placeholders holding down a position, while arguments are values supplied. Declare two parameters, p1 and p2.

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

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.

Examples:

```
// Function to compute the product of p1 and p2
function myFunction(p1, p2) {
  console.log(p1 * p2);
}
myFunction(2, 6);
```

Then, invoke the myFunction() with arguments 2 and 6

Function Return

When JavaScript reaches a return statement, the function will stop executing.

Whatever you return must be assigned to a variable.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

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

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

```
function mySum(num1, num2) {
  let sum;
  sum = num1 + num2;
  return sum;
}
const myTotal = mySum(20, 30);
console.log(myTotal);
```

Default parameter: You can set a default parameter for your function

```
// Default parameter
function people(name = "Janet", job = "Software Developer") {
  console.log("My name is " + name + " " + "I am a " + job);
}
// Invoke the function
people();
```

Functions are re-usable

```
// You can call a function multiple times
const sum = function (number1, number2) {
  return number1 + number2;
};
console.log(sum(3, 3));
console.log(sum(4, 8));
console.log(sum(3, 7));
console.log(sum(2, 14));
console.log(sum(6, 5));
```

Callback Functions

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

In JavaScript, you can also pass a function as an argument to a function. This function that is passed as an argument inside of another function is called a callback function.

```
// function
function greet(name, callback) {
 console.log("Hi" + " " + name);
 callback();
}

// callback function
function callMe() {
```

```
  console.log("I am callback function");
}

// passing function as an argument
greet("Peter", callMe);
```

```
Output:
 Hi Peter
I am callback function
```

In the above program, there are two functions. While calling the greet() function, two arguments (a string value and a function) are passed.

The callMe() function is a callback function.

Note: The callback function is helpful when you have to wait for a result that takes time. For example, the data coming from a server because it takes time for data to arrive.

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**. Arrow functions can be identified by the arrow symbol, => that gives them their name.

Examples:

```
// A function expression
let mult = function (x, y) {
  return x * y;
};

const myMult = mult(2, 5);
console.log(myMult);
```

```
// Arrow function

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

```
};

const myMult = mult(2, 5);
console.log(myMult);
```

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword:

```
const mult = (x, y) => x * y;
const myMult = mult(2, 5);
console.log(myMult);
```

Scope

Scope is an important concept in programming. It refers to where a constant or variable is accessible by the program. Using const and let to declare variables means they are block scoped, so their value only exists inside the block they are declared in.

What is a block scope? A block scope is the area within if, switch conditions or for and while loops. Generally speaking, whenever you see {curly brackets}, it is a block. In ES6, const and let keywords allow developers to declare variables in the block scope, which means those variables, exist only within the corresponding block. There are two types of scope in programs, which are; Global and Local scope.

1. **Global Scope:** Any variable declared outside of a block is said to have global scope. This means it is accessible everywhere in the program.

```
// Global scope

const course = "JavaScript";

function title() {
  console.log(`I love ${course}`);
```

```
}
title();
```

The variable “course” is a global scope variable because it can be accessed anywhere within the program

2. **Local Scope:** This means that any variables defined inside a block using the let or const will only be available inside that block and not be accessible outside of that block.

```
// Local scope

const course = "JavaScript";

if (course === "Java") {
  let book = "Programming";
}
console.log(book);
```

In the above example you will get an error message saying “book is not defined”. This happens because book was defined in a local scope and so therefore it can only be accessed inside the local scope block (if block)

CHAPTER 8

Array

An array is a special variable, which can hold more than one value: It stores list of items, access them, and perform operations on it. Arrays are zero-indexed based, meaning it start from zero.

Examples:

```
const students = ["Joy", "Mary", "Peter", "Ann", "Janet"];
console.log(typeof(students));
```

Why Use Arrays?

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
let car1 = "Toyota";
let car2 = "Volvo";
let car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

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

console.log(cars[0]); // This will return Toyota
console.log(cars[1]); // This will return Volvo
```

```
console.log(cars[2]); // This will return BMW
```

The length property of an array returns the length of an array (the number of array elements)

```
const cars = ["Toyota", "Volvo", "BMW"];
console.log(cars.length); // This will return 3
```

Adding values or items in an array

```
const cars = ["Toyota", "Volvo", "BMW"];
cars[3] = "Ferrari";
console.log(cars);
```

Spread Operator: The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object. It uses the three dots notation (…)

Example:

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
console.log(numbersCombined);

//Output 1,2,3,4,5,6
```

CHAPTER 9

OBJECT

JavaScript object is a non-primitive data-type that allows you to store multiple collections of data.

The syntax to declare an object is:

```
const object_name = {
  key1: value1,
  key2: value2,
};
```

Here, an object object_name is defined. Each member of an object is a key: value pair separated by commas and enclosed in curly braces {}.

Here is an example of a JavaScript object.

```
// object
const student = {
  firstName: "ram",
  age: 10,
};
```

Here, student is an object that stores values such as strings and numbers.

```
// object creation
const person = {
  name: "John",
  age: 20,
};
console.log(typeof person); // object
```

In the above example, name and age are keys, and John and 20 are values respectively.

Object Properties

In JavaScript, "key: value" pairs are called properties. For example,

```
let person = {
  name: "John",
  age: 20,
};
```

Here, name: 'John' and age: 20 are properties.

Accessing Object Properties

You can access the value of a property by using its key.

1. Using dot Notation

Here's the syntax of the dot notation.

```
objectName.key;
```

Example:

```
const person = {
  name: "John",
  age: 20,
};

// accessing property
console.log(person.name); // John
```

2. Using bracket Notation

Here is the syntax of the bracket notation.

```
objectName["propertyName"];
```

Example:

```
const person = {
  name: "John",
  age: 20,
};

// accessing property
console.log(person["name"]); // John
```

JavaScript Nested Objects

An object can also contain another object. For example,

```
// nested object
const student = {
  name: "John",
  age: 20,
  marks: {
    science: 70,
    math: 75,
  },
};

// accessing property of student object
console.log(student.marks); // {science: 70, math: 75}
```

```
// accessing property of marks object
console.log(student.marks.science); // 70
```

In the above example, an object student contains an object value in the mark's property.

JavaScript Object Methods

In JavaScript, an object can also contain a function. A function inside an object is called a method. For example,

```
const person = {
  name: "Sam",
  age: 30,
  // using function as a value
  greet: function () {
    console.log("hello");
  },
};

person.greet(); // hello
```

Here, a function is used as a value for the greet key. That's why we need to use person.greet() instead of person.greet to call the function inside the object.

In JavaScript, objects can also contain functions. For example,

```
// object containing method
const person = {
  name: "John",
  greet: function () {
    console.log("hello");
  },
};
```

In the above example, a person object has two keys (name and greet), which have a string value and a function value, respectively.

Hence basically, the JavaScript method is an object property that has a function value.

JavaScript this Keyword

To access a property of an object from within a method of the same object, you need to use the **this** keyword. Let's consider an example.

```
const person = {
  name: "John",
  age: 30,

  // accessing name property by using this.name
  greet: function () {
    console.log("The name is" + " " + this.name);
  },
};

person.greet();
```

Output

```
The name is John
```

In the above example, a person object is created. It contains properties (name and age) and a method greet.

In the method greet, while accessing a property of an object, this keyword is used.

In order to access the properties of an object, this keyword is used following by. and key.

However, the function inside of an object can access it's variable in a similar way as a normal function would.

For example,

```
const person = {
  name: "John",
  age: 30,
  greet: function () {
    let surname = "Doe";
    console.log("The name is" + " " + this.name + " " + surname);
  },
};

person.greet();
```

Output

```
The name is John Doe
```

Destructuring: The destructuring assignment introduced in ES6 makes it easy to assign array values and object properties to distinct variables.

Example:

```
// Traditional way of assigning object attributes to variables
before ES6
// assigning object attributes to variables before ES6
const person = {
  name: "Sara",
  age: 25,
  gender: "female",
};

let name = person.name;
let age = person.age;
let gender = person.gender;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

```
// Assigning object attributes to variables using destructuring
// assigning object attributes to variables in ES6 using
Destructuring
const person = {
  name: "Sara",
  age: 25,
  gender: "female",
};

// destructuring assignment
let { name, age, gender } = person;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

CHAPTER 10

Properties and Methods

String Properties and Methods

1. **Length:** The length property returns the length of a string:

   Example:

```
let text = "JavaScript";
let length = text.length;
console.log(length);
```

2. **toUpperCase() :** A string is converted to upper case with toUpperCase():

   Example:

```
let name = "ernest";
console.log(name.toUpperCase());
```

3. **toLowerCase() :** A string is converted to lower case with toLowerCase():

   Example:

```
let name = "ERNEST";
console.log(name.toLowerCase());
```

4. **startsWith():** To know the starting character. This returns Boolean (true or false)

Example:

```
let school = "This is our training centre";
console.log(school.startsWith("is"));
```

5. **endsWith():** To know the end character. This returns Boolean (true or false)

Example:

```
let school = "This is our training centre";
console.log(school.endsWith("centre"));
```

6. **indexOf():** The indexOf() method **returns the position of the first occurrence of a value in a string**. The indexOf() method returns -1 if the value is not found. The indexOf() method is case sensitive.

Example:

```
let company = "Welcome to Alabian Solution Limited";
console.log(company.indexOf("Alabian"));
```

```
let company = "Welcome to Alabian Solution Limited";
console.log(company.indexOf("alabian"));

// This will return -1 because alabian is not found
```

7. **lastIndexOf() :** To return the last string of the index you are searching for

Example:

```
let x = "Mr Washington has just arrived Washington";
console.log(x.lastIndexOf("Washington"));
```

8. **slice():** This extracts a part of a string and returns the extracted part in a new string. The method takes 2 parameters: start position, and end position (end not included).

Example:

```
let text = "Apple, Banana, Kiwi";
let part = text.slice(-1);
console.log(part);

// This will return the last letter
```

```
let text = "Apple, Banana, Kiwi";
let part = text.slice(7, 13);
console.log(part);
```

9. **substring():** The substring() is similar to slice(). The difference is that start and end values less than 0 are treated as 0 in substring().

   Example:

```
let str = "Apple, Banana, Kiwi";
let part = str.substring(7, 13);
console.log(part);
```

10. **replace():** The replace() method replaces a specified value with another value in a string:

Example:

```
let x = "I love javascript";
let val = x.replace("javascript", "PHP");
console.log(val);
```

11. **trim() :** The trim() method removes whitespace from both sides of a string.

Example:

```
let x = "Google ";
let val = x.trim();
console.log(val);
```

12. **charAt():** The charAt() method returns the character at a specified index (position) in a string.

Example:

```
let text = "HELLO WORLD";
let char = text.charAt(0);
console.log(char);
```

13. **includes():** The includes() method **determines whether a string contains the given characters within it or not**. This method returns true if the string contains the characters, otherwise, it returns false.

    Example:

```
let str = "Welcome to Qatar.";
let check = str.includes("Qatar");
console.log(check);
```

Number methods

toFixed(): The toFixed() will define the number of characters after the decimal.

Example:

```
let y = 8.0873;
console.log(y.toFixed(2));
```

toString(): The toString() method returns a number as a string.

Example:

```
let y = 8.0873;
let val = y.toString();
console.log(typeof val);
```

parseInt(): parseInt() parses a string and returns a whole number. Spaces are allowed. Only the first number is returned.

Example:

```
console.log(parseInt("-10"));
console.log(parseInt("10"));
console.log(parseInt("10.33"));
```

Array Properties and Methods

1. **Popping and Pushing:** When you work with arrays, it is easy to remove elements and add new elements. This is what popping and pushing is:
   Popping items **out** of an array, or pushing items **into** an array.

pop(): The pop() method removes the last element from an array:

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
     fruits.pop();

Output: ["Banana", "Orange", "Apple"];
```

push(): The push() method adds a new element to an array (at the end):

Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
     fruits.push("Kiwi");

output: ["Banana", "Orange", "Apple", "Mango","Kiwi"];
```

Shifting Elements:

Shifting is equivalent to popping, but working on the first element instead of the last.

shift(): The shift() remove the item at the beginning.

Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();

output: [ "Orange", "Apple", "Mango"];
```

unshift() : The unshift() add element to the beginning of an array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
```

```
output: ["Lemon","Banana", "Orange", "Apple", "Mango"];
```

Array Length: Provides the length of the array.

Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.length);

output:4
```

Merging (Concatenating) Arrays:

The concat() method creates a new array by merging (concatenating) existing arrays:

Example

```
const myGirls = ["Cecilie", "Lone"];
const myBoys = ["Emil", "Tobias", "Linus"];

const myChildren = myGirls.concat(myBoys);
console.log(myChildren);

output: ["Cecilie", "Lone", "Emil", "Tobias", "Linus];
```

Splicing and Slicing Arrays:

The splice() method adds new items to an array. While **the slice()** method slices out a piece of an array.

splice(): The splice() method can be used to add new items to an array:

Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const myFruits = fruits.splice(2, 0, "Lemon", "Kiwi");
console.log(fruits);
```

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (0) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

The splice() method returns an array with the deleted items.

slice(): The slice() method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1 ("Orange"):

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1);
console.log(citrus);
```

Note: The slice() method creates a new array.

The slice() method does not remove any elements from the source array.

sort(): The sort() method sorts an array alphabetically:

Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
console.log(fruits);
```

Note: For numbers the sort will convert the elements to strings. However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Because of this, the sort() method will produce incorrect result when sorting numbers.

```
const numbers = [14, 18, 5, 9, 10, 23, 25];
numbers.sort();
console.log(numbers);
```

Reversing an Array:

The reverse() method reverses the elements in an array. You can use it to sort an array in descending order:
Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.reverse();
console.log(fruits);
```

Include in Array:

The includes() method determines whether an ***array includes*** a certain value among its entries, returning true or false as appropriate.

Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const myFruits = fruits.includes("Orange");
console.log(myFruits);
```

Join(): The ***join*()** ***method*** returns an ***array*** as a string. The ***join*()** ***method*** does not change the original ***array***. Any separator can be specified. The default is comma.

Example:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const myFruits = fruits.join();
console.log(myFruits);
```

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const myFruits = fruits.join(", ");
console.log(myFruits);
```

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const myFruits = fruits.join(" - ");
console.log(myFruits);
```

Array Iteration

Array iteration methods operate on every array item.

forEach(): The forEach() method calls a function (a callback function) once for each array element. The forEach() does not return a new array.

Example:

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

cars.forEach(function (car) {
  console.log("I love " + car);
});
```

map (): The map() method creates a new array by performing a function on each array element. The map() method does not change the original array.

Example

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

const myCar = cars.map(function (car) {
  return "I love " + car;
});
console.log(myCar);
```

filter(): The filter() is an inbuilt method, this method creates a new array with elements that follow or pass the given criteria and condition.

Example:

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

const filteredCar = cars.filter(function (car) {
  return car === "Ferrari";
});

console.log(filteredCar);
```

find(): To find elements in an array you use find(). It will return the first element being searched in an array.

Example:

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

const findCar = cars.find(function (car) {
  return car === "Toyota";
});
console.log(findCar);
```

every(): The every() method checks if all array values pass a test. It returns truthy if the callback returns true.

Example:

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

const everyCar = cars.every(function (car) {
  return car === "Volvo";
});
console.log(everyCar);

// This will return true
```

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

const everyCar = cars.every(function (car) {
  return car === "Volvo";
});
console.log(everyCar);

// This will return false as we do not only have "Volvo" in the
array
```

some(): The some() method checks if some array values pass a test. It returns truthy if the callback returns true.

Example:

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

const someCar = cars.some(function (car) {
  return car === "Volvo";
});
console.log(someCar);

// This will return true as at least one item "Volvo" is
available in the array
```

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

const someCar = cars.some(function (car) {
  return car === "Bughatti";
});
console.log(someCar);

// This will return false as no item as "Bughatti" exist in the
array
```

indexOf(): The indexOf() method searches an array for an element value and returns its position.

Example:

```
const cars = ["Toyota", "Benz", "Ferrari", "Volvo"];

const indexCar = cars.indexOf("Benz");
console.log(indexCar);

// This will return the index of a selected item
```

CHAPTER 11

CLASSES

Classes: Classes are one of the features introduced in the ES6 version of JavaScript. A class is a blueprint for the object. You can create an object from the class.

```
// creating a class

class Person {
  constructor(name) {
    this.name = name;
  }
}
```

The **class** keyword is used to create a class. The properties are assigned in a constructor function.

constructor

The **constructor** method is a special method of a **class** for creating and initializing an object instance of that class.

A constructor enables you to provide any custom initialization that must be done before any other methods can be called on an instantiated object.

In JavaScript, the **this** keyword refers to an **object.**

The **new** keyword is used in JavaScript to create a object from a constructor function. The new keyword has to be placed before the constructor function call and will do the following things: Creates a new object. Sets the prototype of this object to the constructor function's prototype property.

```
// creating a class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```
// creating an object
const person1 = new Person("John");
const person2 = new Person("Jack");

console.log(person1.name); // John
console.log(person2.name); // Jack
```

Here, person1 and person2 are objects of Person class.

Note: The constructor() method inside a class gets called automatically each time an object is created.

JavaScript Class Methods

It is easy to define methods in the JavaScript class. You simply give the name of the method followed by ().

For example,

```
class Person {
  constructor(name) {
    this.name = name;
  }

  // defining method
  greet() {
    console.log(`Hello ${this.name}`);
  }
}

let person1 = new Person("John");

// accessing property
console.log(person1.name); // John

// accessing method
person1.greet(); // Hello John
```

Note: To access the method of an object, you need to call the method using its name followed by ().

Getters and Setters

In JavaScript, getter methods get the value of an object and setter methods set the value of an object.

JavaScript classes may include getters and setters. You use the get keyword for getter methods and set for setter methods.

For example:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  // getter
  get personName() {
    return this.name;
  }

  // setter
  set personName(x) {
    this.name = x;
  }
}

let person1 = new Person("Jack");
console.log(person1.name); // Jack

// changing the value of name property
person1.personName = "Sarah";
console.log(person1.name); // Sarah
```

Class inheritance

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more.

Using class inheritance, a class can inherit all the methods and properties of another class.

Inheritance is a useful feature that allows code reusability.

To use class inheritance, you use the extends keyword.

For example:

```
// parent class
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello ${this.name}`);
  }
}

// inheriting parent class
class Student extends Person {}

let student1 = new Student("Jack");
student1.greet(); //Hello Jack
```

In the above example, the Student class inherits all the methods and properties of the Person class. Hence, the Student class will now have the name property and the greet() method.

JavaScript Super() Keyword

The super keyword used inside a child class denotes its parent class.

For example,

```
// parent class
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello ${this.name}`);
  }
}
```

```
// inheriting parent class
class Student extends Person {
  constructor(name) {
    console.log("Creating student class");

    // call the super class constructor and pass in the name
parameter
    super(name);
  }
}

let student1 = new Student("Jack");
student1.greet();
```

Here, super inside Student class refers to the Person class. Hence, when the constructor of Student class is called, it also calls the constructor of the Person class which assigns a name property to it.

CHAPTER 12

Date Objects and Math Objects

Date object

Date objects hold information about dates and times. A constructor function is used to create a new date object using the new operator. Get today's time

```
const today = new Date();
```

If an argument is not supplied, the date will default to the current date and time. The parmeters that can be provided are as follows:

New Date(year, month, day, hour, minutes, seconds, milliseconds).

Getting Methods

¬ getDate(): This is used to find the day of the week. It returns a number starting from zero(0) i.e Sunday will return 0, Monday will return 1 etc.

¬ getDate(): This returns the day of the month.

¬ getMonth(): tis returns an integer starting from zero(0) i.e January is 0, Febuary is 1 etc

¬ getFullYear(): This returns the year in 4 digits e.g 2018.

¬ We have the following methods; getHour(), getMinutes(), getSeconds(), getMilliseconds().

Math Object

The **JavaScript math** object provides several constants and methods to perform mathematical operation. Unlike date object, it doesn't have constructors.

The JavaScript Math object allows you to perform mathematical tasks on number.

Math methods

Mathematics plays an integral part in computer science and coding. Programmers use mathematical methods and expressions to perform calculations for all sorts of different

reasons during development. Luckily, JavaScript provides various built-in methods that can make your life a whole lot easier.

1. **Math.abs() :** The **abs()** method returns the absolute value of a number.
   Example:

```
let num1 = 32;
let num2 = -13;
let num3 = -345;
let num4 = 4.76;
let num5 = 0;
console.log(Math.abs(num1));
console.log(Math.abs(num2));
console.log(Math.abs(num3));
console.log(Math.abs(num4));
```

2. **Math.round():** The **round()** method returns the value of a number rounded to the nearest integer.
   Example:

```
let num1 = 34.5;
let num2 = 54.234;
let num3 = 7.0001;
let num4 = 867.1;

console.log(Math.round(num1));
console.log(Math.round(num2));
console.log(Math.round(num3));
console.log(Math.round(num4));
```

3. **Math.ceil():** The **ceil()** method returns the next integer greater than or equal to a given number.
   Example:

```
let num1 = 34.5;
let num2 = 54.234;
```

```
let num3 = 7.0001;
let num4 = 867.1;

console.log(Math.ceil(num1));
console.log(Math.ceil(num2));
console.log(Math.ceil(num3));
console.log(Math.ceil(num4));
```

4. **Math.floor():** The **floor()** method returns the next integer less than or equal to a given number.
   Example:

```
let num1 = 34.5;
let num2 = 54.234;
let num3 = 7.0001;
let num4 = 867.1;

console.log(Math.floor(num1));
console.log(Math.floor(num2));
console.log(Math.floor(num3));
console.log(Math.floor(num4));
```

5. Math.pow(x, y): Math.pow(x, y) returns the value of x to the power of y:
   Example:

```
let x = 8;
let y = 2;
console.log(Math.pow(x, y));
```

6. **Math.sqrt(x):** The Math.sqrt(x) returns the square root of x:
   Example:

```
let x = 64;
console.log(Math.sqrt(x));
```

7. **Math.max():** The Math.max() can be used to find the highest value in a list of arguments:

   Example:

```
console.log(Math.max(3, 9, 1, 5, 6));
```

8. **Math.min():** The Math.min() can be used to find the lowest value in a list of arguments.
   Example:

```
console.log(Math.min(3, 9, 1, 5, 6));
```

9. **Math.random():** The Math.random() returns a random number between 0 (inclusive), and 1 (exclusive):
   Example:

```
console.log(Math.random());
```

CHAPTER 13

Local Storage in JavaScript

Local storage continues to store data after the browser is closed.

Local storage is a property of the window object which simply represents an open window in the browser.

Methods used for Local storage

1. **setItem() –** To save item in a local storage. It adds key and value to local storage.

2. **getItem() –** To get items from localStorage, use the getItem() method. getItem() allows you to access the data stored in the browser's localStorage object.

3. **removeItem() –** Remove an item by key from local storage.

4. **Clear() –** Clear all items in the local storage.

5. **Key() –** Pass a number to retrieve key of a local storage.

SetItem()

SetItem() is used to save data to local storage.

Example:

```
// Saving data to localStorage
localStorage.setItem("firstName", "Peter");
```

To see the item in your local storage, you right click the browser, click on "inspect" then click on "Application" on the console, then click on "Local Storage" to view the stored item.

Note: If the data to be stored is an array or object it has to be converted to a string by using the **JSON.stringify()**

```
// Convert an array to a string before saving into localStorage
const students = ["John", "Mary", "Michael"];
localStorage.setItem("myStudents", JSON.stringify(students));
```

```
// Convert an object to a string before saving into localStorage
const person = {
  firstName: "Isibor Ernest",
  job: "Software developer",
  country: "Nigeria",
};
localStorage.setItem("user", JSON.stringify(person));
```

GetItem()

GetItem() is used to get item from local storage. getItem() accepts only one parameter, which is the key, and returns the value as a string.

```
// Getting item from LocalStorage
console.log(localStorage.getItem("firstName"));
```

```
// Convert an object to a string before saving into localStorage
const person = {
  firstName: "Isibor Ernest",
  job: "Software developer",
  country: "Nigeria",
};
localStorage.setItem("user", JSON.stringify(person));
```

To use this value, you would have to convert it back to an object.

To do this, we make use of the JSON.parse() method, which converts a JSON string into a JavaScript object.

```
JSON.parse(window.localStorage.getItem('person'));
```

RemoveItem()

Remove an item by key from local storage.

When passed a key name, the `removeItem()` method removes that key from the storage if it exists. If there is no item associated with the given key, this method will do nothing.

```
// Removing item from LocalStorage
console.log(localStorage.removeItem("firstName"));
```

Clear()

Clear all items in local storage.

```
// Clear item from LocalStorage
console.log(localStorage.clear());
```

Limitation of Local Storage

1. Do not store sensitive user information in local storage

2. It is not a substitute for server-based database as information is only stored in the browser.

3. Local storage is limited to 5MB across all major browsers.

4. Local storage is quite insecure as it has no form of data protection and can be accessed by any code on your web page.

5. Local storage is synchronous, meaning each operation call would only executes after they are called. 6. Local storage can only store strings. You will need to convert arrays, objects into strings to store items in the local storage.

CHAPTER 14

Introduction to Document Object Model (DOM)

What is the DOM?

DOM stands for Document Object Model. It is a programming interface that allows us to create, change, or remove elements from the document. We can also add events to these elements to make our page more dynamic.

The DOM views an HTML document as a tree of nodes. A node represents an HTML element.

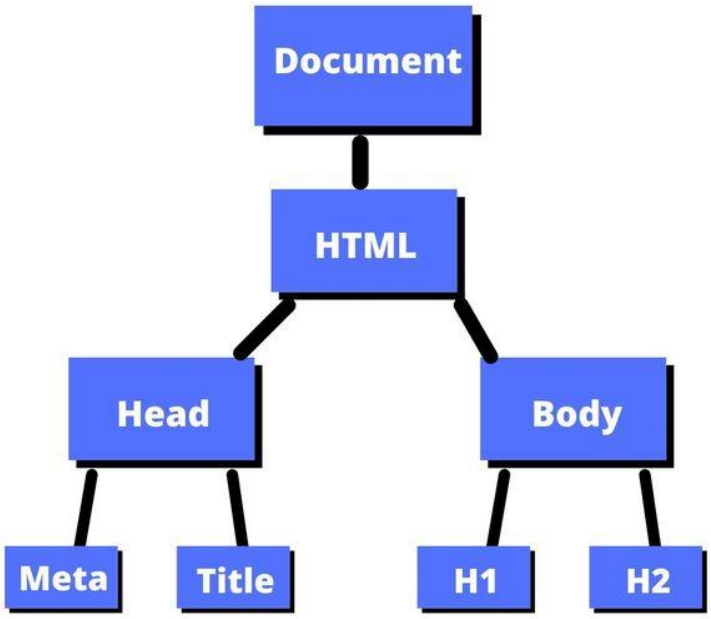
Let's take a look at this HTML code to better understand the DOM tree structure.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <h1>DOM tree structure</h1>
    <h2>Learn about the DOM</h2>
</body>

</html>
```

Our document is called the root node and contains one child node which is the <html> element. The <html> element contains two children which are the <head> and <body> elements.

Both the <head> and <body> elements have children of their own.

Here is another way to visualize this tree of nodes.



We can access these elements in the document and make changes to them using JavaScript.

Let's take a look at how we can work with the DOM using JavaScript.

How to Select Elements in the Document

There are a few different methods for selecting an element in the HTML document. They are:

1. getElementById()
2. getElementsByTagName()
3. getElementsByClassName()
4. querySelector()
5. querySelectorAll()

getElementById(): The **getElementById()** is a JavaScript function that lets you grab an HTML element, by its id, and perform an action on it. The name of the id is passed as a parameter, and the corresponding element is the return value.

Syntax:

```
const element = document.getElementById(id);
```

Example:

```
<h1 id="home">Hello world</h1>;
```

```
// This showcase the h1 element on the console
const heading = document.getElementById("home");
console.log(heading);
```

getElementsByTagName(): The getElementsByTagName() method returns a HTML collection of an elements's child elements with the specified tag name, as a NodeList object. getElementsByTagName() returns a node-list. A node-list is an array-like object. It only works for the index and length property but not for array methods. Syntax:

```
const element = document.getElementsByTagName('p')
```

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <p>Samsumg</p>
    <p>Gionee</p>
    <p>Nokia</p>
    <p>Techno</p>

</body>

</html>
```

```
//This will display an array of HTMLCollection
const para = document.getElementsByTagName("p");
console.log(para);
```

getElementsByClassName(): This method will return every element that has a class name that is supplied as an argument. gwtElementsByClassName() returns a HTML collection or a node-list. A node-list is an array-like object. It only works for the index and length property but not for array methods. You can access individual item in a node-list. With the advent of ES6, you can use the for of () to loop through.

Syntax:

```
const element = document.getElementsByClassName("home");
```

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <h1 class="home">Book one</h1>
    <h1 class="home">Book two</h1>
    <h1 class="home">Book three</h1>

</body>

</html>
```

```
// This will display an array of HTMLCollection
const heading = document.getElementsByClassName("home");
console.log(heading);
```

```
// Since it is an array of HTMLCollection you can get a single
element
const heading = document.getElementsByClassName("home");
console.log(heading[2]);
```

querySelector(): The querySelector() method returns the **first** element that matches a CSS selector. To return **all** matches (not only the first), use the querySelectorAll() instead.

It uses the pound (#) symbol to reference an id while it uses the dot (.) symbol to reference a class.

Syntax:

```
const element = document.querySelector("#home");
```

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <h1 id="home">Hello world</h1>
    <h1 id="home">Welcome home</h1>
    <h4 class="myName">My name is John Doe</h4>
    <script src="hello.js"></script>
</body>

</html>
```

```
// This will display only the first element with the home id
const home = document.querySelector("#home");
console.log(home);
```

querySelectorAll(): The querySelectorAll() method returns all elements that matches a CSS selector(s). The querySelectorAll() method returns a NodeList.

It uses the pound (#) symbol to reference an id while it uses the dot (.) symbol to reference a class.

Syntax:

```
const element = document.querySelectorAll(".phone");
```

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <p class="phone">Samsumg</p>
    <p class="phone">Gionee</p>
    <p class="phone">Nokia</p>
    <p class="phone">Techno</p>

    <script src="hello.js"></script>
</body>

</html>
```

```
// To get the array of NodeList
const phone = document.querySelectorAll(".phone");
console.log(phone);
```

You can as well loop through:

```
// You can loop through the array of NodeList
const phone = document.querySelectorAll(".phone");
phone.forEach(function (item) {
  console.log(item);
});
```

CHAPTER 15

Traversing the DOM

DOM traversal (also called walking or navigating the DOM) is **the act of selecting nodes in the DOM tree from other nodes**. You're probably already familiar with several methods for accessing elements in the DOM tree by their id, class, or tag name. You can use methods like document.

Using the children or childNodes Properties

childNodes returns all direct child nodes (not just child elements). If you are only interested in child elements, say list items only, use the **children** property.

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <ul id="result">
        <li>Apple</li>
        <li>Orange</li>
        <li>Banana</li>
        <li>Pawpaw</li>
        <li>Mango</li>
    </ul>

    <script src="hello.js"></script>
</body>
```

```
</html>
```

```
const result = document.querySelector("#result");
const allChildren = result.childNodes;
console.log(allChildren);
```

The **children** property selects all child elements that are directly under a given element. Here's an example of the **children** property in action:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <ul id="result">
        <li>Apple</li>
        <li>Orange</li>
        <li>Banana</li>
        <li>Pawpaw</li>
        <li>Mango</li>
    </ul>

    <script src="hello.js"></script>
</body>

</html>
```

```
const result = document.querySelector("#result");
const children = result.children;
console.log(children);
```

lastChild and firstChild Properties

As their names suggest, the **lastChild** and **firstChild** properties return an element's last and first child nodes.

parentElement or parentNode

Both **parentElement** or **parentNode** properties let you select the selected element's parent node one level up. The critical difference is that **parentElement** only chooses the parent node that is an element. On the other hand, **parentNode** can select a parent regardless of whether it's an element or a different node type.

Example:

```
const result = document.querySelector("#result");
const firstChild = result.firstChild;
console.log(firstChild);
```

```
const result = document.querySelector("#result");
const lastChild = result.lastChild;
console.log(lastChild);
```

nextElementSibling or previousElementSibling

nextElementSibling to select the following sibling element
and **previousElementSibling** to select the previous sibling.

Example of nextSibling

```
// nextSibling
const first = document.querySelector(".first");
const second = first.nextSibling;
console.log(second);

// This will return a text node which is whitespace
```

```
// nextSibling
const first = document.querySelector(".first");
const second = first.nextSibling.nextSibling;
console.log(second);

// This will return the next sibling
```

Example of previousSibling

```
// previousSibling
const last = document.querySelector(".last");
const second = last.previousSibling;
console.log(second);

// This will return a text node which is whitespace
```

```
// previousSibling
const last = document.querySelector(".last");
const second = last.previousSibling.previousSibling;
console.log(second);

// This will return the previous sibling
```

CHAPTER 16

DOM Styling – Applying Styling With CSS

The style property returns a CSS Style Declaration object, which represents an element's style attribute. The style property is used to get or set a specific style of an element using different CSS properties.

Note: It is not possible to set styles by assigning a string to the style property, e.g. element.style = "color: red;". To set the style of an element, append a "CSS" property to style and specify a value, like this:

```
element.style.backgroundColor = "red";
//set the background color of an element to red
```

Styling DOM Elements in JavaScript

You can also apply style on HTML elements to change the visual presentation of HTML documents dynamically using JavaScript. You can set almost all the styles for the elements like, fonts, colors, margins, borders, background images, text alignment, width and height, position, and so on.

Applying Styles on Elements.

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
```

```
    <p id="intro">This is a paragraph.</p>
    <p>This is another paragraph.</p>
    <script src="hello.js"></script>
</body>

</html>
```

```
// Selecting element
let elem = document.getElementById("intro");
// Appling styles on element
elem.style.color = "blue";
elem.style.fontSize = "18px";
elem.style.fontWeight = "bold";
```

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <div id="result">
        <h1>Hello world</h1>
        <div class="second">
            <h2>Second heading</h2>
        </div>
    </div>

    <script src="hello.js"></script>
</body>
```

```
</html>
```

```
const heading = document.querySelector("h2");
const parent = (heading.parentElement.style.color = "red");
console.log(parent);
```

Naming Conventions of CSS Properties in JavaScript

Many CSS properties, such as font-size, background-image, text-decoration, etc. contain hyphens (-) in their names. Since, in JavaScript hyphen is a reserved operator and it is interpreted as a minus sign, so it is not possible to write an expression, like: elem.style.font-size.

Therefore, in JavaScript, the CSS property names that contain one or more hyphens are converted to intercapitalized style word. It is done by removing the hyphens and capitalizing the letter immediately following each hyphen, thus the CSS property font-size becomes the DOM property fontSize, border-left-style becomes borderLeftStyle, and so on.

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <p id="intro">This is a paragraph.</p>
    <p>This is another paragraph.</p>
```

```
    <script src="hello.js"></script>
</body>

</html>
```

```
// Selecting element
let elem = document.getElementById("intro");
// Appling styles on element
elem.style.backgroundColor = "yellow";
elem.style.fontSize = "18px";
elem.style.fontWeight = "bold";
```

CHAPTER 17

DOM Manipulation

DOM manipulation is the interaction of the JavaScript DOM API to modify or change the HTML document. With DOM manipulation, you can create, modify, style, or delete elements without a refresh. It also promotes user interactivity with browsers.

Adding CSS Classes to Elements

You can also get or set CSS classes to the HTML elements using the className property.

Since, class is a reserved word in JavaScript, so JavaScript uses the className property to refer the value of the HTML class attribute.

Example

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <p id="intro">This is a paragraph.</p>

    <script src="hello.js"></script>
</body>

</html>


// Selecting element
```

Example

```
let elem = document.getElementById("intro");
elem.className = "note";
let myNote = document.querySelector(".note");
// Appling styles on element
myNote.style.backgroundColor = "pink";
myNote.style.fontSize = "48px";
myNote.style.fontWeight = "bold";
```

ClassName and ClassList

Using "classList", you can add or remove a class without affecting any others the element may have. But if you assign "className", it will wipe out any existing classes while adding the new one.

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <p id="intro">This is a paragraph.</p>
    <p class="good">Good girl gone bad</p>
    <p class="bad">Bad girl gone good</p>

    <script src="hello.js"></script>
</body>

</html>
```

```
const good = document.querySelector(".good");
const way = (good.className = "text");
console.log(good);

// The new className will replaced the previous class
```

There is even better way to work with CSS classes. You can use the classList property to get, set or remove CSS classes easily from an element.

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <div id="info" class="disabled">Something very
important!</div>

    <script src="hello.js"></script>
</body>

</html>
```

```
// Selecting element
let elem = document.getElementById("info");
elem.classList.add("hide"); // Add a new class
elem.classList.add("note", "highlight"); // Add multiple classes
elem.classList.remove("hide"); // Remove a class
```

```
elem.classList.remove("disabled", "note"); // Remove multiple
classes
elem.classList.toggle("visible"); // If class exists remove it,
if not add it
console.log(elem);
```

contains

The ***contains***() method is used to determines whether the collection ***contains*** a given item or not.

```
// Determine if class exist
if (elem.classList.contains("highlight")) {
  alert("The specified class exists on the element.");
}
```

Creating Elements

in JavaScript You use the createElement () to create element

```
// Create HTML elements in JavaScript
const text = document.createElement("div");
console.log(text);
```

Creating TextNode

The ***createTextNode***() method creates a text node.

```
// Create HTML elements in JavaScript
const text = document.createElement("div");
const bodyDiv = document.createTextNode("A simple website");
console.log(bodyDiv);
```

Append Child

appendChild() – To append a child to a parent element

```
// Create HTML elements in JavaScript
const text = document.createElement("div");
const bodyDiv = document.createTextNode("A simple website");
text.appendChild(bodyDiv);
document.body.appendChild(text);
```

```
// Create HTML elements in JavaScript
const text = document.createElement("div");
const bodyDiv = document.createTextNode("A simple website");
// Add CSS dynamically
text.classList.add("myColor");
text.appendChild(bodyDiv);
document.body.appendChild(text);
```

Example:

```
// Create HTML elements in JavaScript
const newLink = document.createElement("a");
console.log(newLink);
```

To add an id element

```
// Create HTML elements in JavaScript
const newLink = document.createElement('a')
// Add id element
newLink.id = "myLink";
console.log(newLink);
```

To add href attribute to an id link

```
// Create HTML elements in JavaScript
const newLink = document.createElement("a");
// Add href to the link
newLink.id = "myLink";
newLink.href = "https://www.facebook.com";
console.log(newLink);
```

To add a class element

```
// Create HTML elements in JavaScript
const newLink = document.createElement("a");
// Add class element
newLink.className = "myClassLink";
console.log(newLink);
```

To add content to the link

```
// Create HTML elements in JavaScript
const newLink = document.createElement('a')
// Add href to the link
newLink.id = "myLink";
newLink.href = "https://www.facebook.com";
// Add content to the link
newLink.textContent = "facebook";
```

AppendChild

AppendChild() – To append a child to a parent element

```
// Create HTML elements in JavaScript
const newLink = document.createElement("a");
const myBody = document.body;
// Add href to the link
```

```
newLink.id = "myLink";
newLink.href = "https://www.facebook.com";
// Add content to the link
newLink.textContent = "facebook";
// Appending a child element (newLink) to a parent element
(myBody)
myBody.appendChild(newLink);
console.log(newLink);
```

Remove

Remove elements from DOM – remove()

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>
<style>
    .myColor {
        color: blue;
    }
</style>

<body>
    <div id="info" class="disabled">Something very
important!</div>

    <div id="result">Remove me</div>

    <div>
        <h1>Nigeria</h1>
        <h2>Ghana</h2>
```

```
        <h3>Togo</h3>
    </div>


    <script src="hello.js"></script>
</body>

</html>
```

```
const result = document.querySelector("#result");
const heading = result.remove();
```

removeChild

removeChild():

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>
<style>
    .myColor {
        color: blue;
    }
</style>

<body>
    <div id="info" class="disabled">Something very
important!</div>
```

```
    <div id="result">
        <h1>Nigeria</h1>
        <h2>Ghana</h2>
        <h3>Togo</h3>
    </div>


    <script src="hello.js"></script>
</body>

</html>
```

```
const result = document.querySelector("#result");
const heading = result.querySelector("h1");
result.removeChild(heading);
```

setAttribute

setAttribute() – To set attribute to elements in DOM

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>
<style>
    .myColor {
        color: blue;
    }
</style>
```

```
<body>
    <div id="info" class="disabled">Something very
important!</div>
    <h3 class="good">Home</h3>


    <script src="hello.js"></script>
</body>

</html>
```

```
// To set Attribute to an element
const newElement = document.createElement("h3");
newElement.setAttribute("class", "wonders");
newElement.setAttribute("id", "go");
console.log(newElement);
```

removeAttribute

removeAttribute() – To remove attribute from elements in DOM

```
// To set Attribute to an element
const newElement = document.createElement("h3");
newElement.setAttribute("class", "wonders");
newElement.setAttribute("id", "go");
// To remove attribute
newElement.removeAttribute("class");
console.log(newElement);
```

InnerHTML, InnerText and TextContent

1. **innerText** property returns the content of all elements, except for `<script>` and `<style>` elements. The returned content is visible plain text without any formatting, similar to highlighting text and then copying and pasting it. What you see is what you get.

Example:

```
const displayText = document.createElement("div");
displayText.innerText = "<p>I love programming<p>";
displayText.style.color = "red";
document.body.appendChild(displayText);
```

2. **innerHTML** property returns the string inside our `div` and the HTML (or XML) markup contained within our string, including any spacing, line breaks and formatting irregularities.

   Example:

```
const displayText = document.createElement("div");
displayText.innerHTML = "<b>I love programming<b>";
displayText.style.color = "red";
document.body.appendChild(displayText);
```

3. **textContent** property returns the raw content with styling inside of all elements, but excludes the HTML element tags.
   Example:

```
const displayText = document.createElement("div");
displayText.textContent = "<b>I love programming<b>";
displayText.style.color = "red";
document.body.appendChild(displayText);
```

CHAPTER 19

Events

What is an Event?

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

Here are some of the few events:

onclick Event Type

This is the most frequently used event type which occurs when a user clicks the left button of his mouse. You can put your validation, warning etc., against this event type.

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <p>Click the following button and see result</p>
    <form>
        <button type="button" onclick="sayHello()">Say
Hello</button>
```

```
    </form>
    <script src="hello.js"></script>
</body>

</html>
```

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

onsubmit Event Type

onsubmit is an event that occurs when you try to submit a form. You can put your form validation against this event type.

onmouseover and onmouseout

These two event types will help you create nice effects with images or even with text as well. The **onmouseover** event triggers when you bring your mouse over any element and the **onmouseout** triggers when you move your mouse out from that element. Try the following example.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
```

```
    <p>Bring your mouse inside the division to see the
result:</p>
    <img src="images/backgroung.png "
        width="250px" height="250px"
onmouseover="over()"  alt="">
    <div onmouseout="out()">
        <h2> This is inside the division </h2>
    </div>
    <script src="hello.js"></script>
</body>

</html>
```

```
function over() {
  document.write("Mouse Over");
}
function out() {
  document.write("Mouse Out");
}
```

There are other events which you can explore on but we will not cover it in this manual.

Event Listeners

An event listener is a function that initiates a predefined process if a specific event occurs. So, an event listener “listens” for an action, then calls a function that performs a related task. This event can take one of many forms. Common examples include mouse events, keyboard events, and window events.

Many web applications rely on some form of event to carry out their functions. At some point, a human interacts with their interface, which generates an event. These human-driven events typically rely on a peripheral device, such as a mouse or keyboard.

When a device creates an event, the program can listen for it, to know when to carry out specific behavior. In this tutorial, you'll learn how to listen for events using JavaScript.

Creating an Event Listener

You can listen for events on any element in the DOM. JavaScript has an addEventListener() function that you can call on any element on a web page. The addEventListener() function is a method of the EventTarget interface.

The addEventListener() function has the following basic structure:

```
element.addEventListener("event", functionToExecute);
```

Where:

- the element can represent any HTML tag (from a button to a paragraph)
- the "event" is a string naming a specific, recognized action
- the functionToExecute is a reference to an existing function

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <h1 class="hello">Hello world!</h1>
    <button id="addClick">Click me</button>

    <script src="hello.js"></script>
</body>
```

```
</html>
```

```
// Adding an event listener to your code
const hello = document.getElementById("hello");
const addClick = document.getElementById("addClick");
// Add event listener to the button
addClick.addEventListener("click", function () {
  alert("You have just clicked me!");
});
```

This will trigger an action “You have just clicked me” once the button is clicked

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>
<style>
    .reveal {
        color: white;
        background-color: red;
        padding: 5px;
    }
</style>

<body>
    <div>
        <h1 class="msg">Welcome to JavaScript training</h1>
        <button type="button" id="btn">Toggle Message</button>
    </div>
```

```
    <script src="hello.js"></script>
</body>

</html>
```

```
// Adding an event listener to your code
const btn = document.getElementById("btn");
const msg = document.querySelector(".msg");

btn.addEventListener("click", function () {
  if (msg.classList.contains("msg")) {
    msg.classList.remove("msg");
    msg.classList.add("reveal");
    console.log(msg);
  } else {
    console.log("NO");
  }
});
```

You can as well replace the function() with a variable name and declare it somewhere else in your code.

Example:

```
// Adding an event listener to your code
const btn = document.getElementById("btn");
const msg = document.querySelector(".msg");

btn.addEventListener("click", changeMe);

// Create a function
function changeMe() {
  if (msg.classList.contains("msg")) {
    msg.classList.toggle("reveal");
  } else {
    console.log("NO");
```

```
  }
}
```

Using the event delegation

You can get the events properties.

```
// Adding an event listener to your code
const btn = document.getElementById("btn");
const msg = document.querySelector(".msg");

// Add event listener to the button
btn.addEventListener("click", myClick);
// Write the function here
function myClick(e) {
  console.log(e);
}
```

```
// Adding an event listener to your code
const btn = document.getElementById("btn");
const msg = document.querySelector(".msg");

// Add event listener to the button
btn.addEventListener("click", myClick);
// Write the function here
function myClick(e) {
  console.log(e);
  console.log(e.target.id);
  console.log((e.target.innerText = "Send"));
}
```

CHAPTER 20

Form Events

A form event is fired when a form control receives or loses focus or when the user modifies a form control value such as by typing text in a text input, select any option in a select box etc.

An action is triggered whenever a user clicks or submit a form. A key form event for submitting is the “submit” event

Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <div>
        <h1>Form event</h1>
        <form action="" method="post" id="myForm">
            <label for="Username">Username</label>
            <input type="text" name="username" id="username"
placeholder="Enter
username"> <br><br>
            <label for="Password">Password</label>
            <input type="password" name="password" id="password"
placeholder="Enter
password"><br><br>
            <button type="submit" id="btn">Submit</button>
        </form>
    </div>
```

```
    <script src="hello.js"></script>
</body>

</html>
```

If you noticed, on the console the message blinks and did not display result permanently, this is as a result of the browser that automatically reloads a page whenever a form is submitted. To prevent this action you will need to insert the “e.preventDefault()” method to stop the browser from reloading your form.

e.preventDefault() stops the browser from reloading after a form is submitted. Inshort e.preventDefault() prevents default behavior.

```
// Form events
const myForm = document.getElementById("myForm");
const username = document.getElementById("username");
const password = document.getElementById("password");
myForm.addEventListener("submit", formEvent);
function formEvent(e) {
  e.preventDefault();
  console.log("Form submitted");
}
```

To get what the user’s typed in the form, you use the .value property.

Example

```
// Form events
const myForm = document.getElementById("myForm");
const username = document.getElementById("username");
const password = document.getElementById("password");
myForm.addEventListener("submit", formEvent);
function formEvent(e) {
  e.preventDefault();
  console.log(username.value);
  console.log(password.value);
  console.log("Form submitted");
}
```

Example:

Let create a To-do app

Index.html file

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="preconnect" href="https://fonts.googleapis.com" />
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
    <link
      href="https://fonts.googleapis.com/css2?family=DM+Sans:wght@400;500;700&dis
play=swap"
      rel="stylesheet"
    />
    <title>Document</title>
    <style>
      body {
        font-family: "DM Sans", sans-serif;
        background-color: rgb(9, 43, 75);
        height: 90vh;
        box-sizing: border-box;
      }
      .todo-text {
        text-align: center;
      }
      .todo-text h2 {
        color: white;
      }
      #form {
        display: flex;
        justify-content: center;
        margin: 60px 0px;
      }
      #todo {
        padding: 10px;
        margin: 0px 20px;
        /* border-radius: 10px; */
        /* border: none; */
        /* box-shadow: none; */
        width: 280px;
        /* height: 90px; */
```

```
    }
    #form button {
      border-radius: 10px;
      border: none;
      padding: 0 25px;
      background-color: #27649c;
      color: white;
      font-size: 18px;
    }
    .todoText {
      list-style-type: none;
      display: flex;
      flex-direction: column;
      align-items: center;
    }
    .todoText li {
      background-color: #f8f9fa;
      border-radius: 10px;
      box-shadow: 1px 1px 10px black;
      width: 300px;
      /* height: 45px; */
      padding: 20px;
      margin: 10px 0px;
    }
    .todo-app {
      padding: 10px 10px;
    }
  </style>
</head>
<body>
  <div class="todo-app">
    <div class="todo-text">
      <h2>TODO APP</h2>
      <img src="images.jpg" width="90" />
    </div>
    <form id="form">
      <input type="text" id="todo" />
      <button class="todo_submit">Submit</button>
    </form>
    <div>
      <h2 style="text-align: center; color: white">Todo List</h2>
      <div class="todo-item">
        <ul class="todoText">
          <!-- <li>Rice</li> -->
        </ul>
```

```
        </div>
      </div>
    </div>

    <script src="Todo.js"></script>
    <script src="https://unpkg.com/sweetalert/dist/sweetalert.min.js"></script>
  </body>
</html>
```

Let create Todo.js file

```
const todoInput = document.getElementById("todo");
const btn = document.querySelector(".todo_submit");
const todoText = document.querySelector(".todoText");
const myForm = document.getElementById("form");
```

Let add a click event listener to myForm

```
myForm.addEventListener("submit", todoApp);
```

Note: what we want to achieve is that anytime we click on submit button it should add a new list to the parent list, that is the reason why we used appendChild

```
const todoInput = document.getElementById("todo");
const btn = document.querySelector(".todo_submit");
const todoText = document.querySelector(".todoText");
const myForm = document.getElementById("form");

myForm.addEventListener("submit", todoApp);

let output;
function todoApp(e) {
  e.preventDefault();
  // creating li element
  const li = document.createElement("li");
  //   console.log(todoText);
```

```
  if (todoInput.value.length == "0") {
    swal("Please enter a text", "", "error");
  } else {
    output = todoInput.value;
    li.append(output);
    todoText.appendChild(li);
  }
}
// console.log(li);
```

Let add data to localstorage

Note: If the data to be stored is an array or object it has to be converted to a string by using the **JSON.stringify()**

Form Validation

Form validation normally used to occur at the server, after the client had entered all the necessary data and then pressed the submit button. If the data entered by the client was incorrect or was simply missing, the server would have to send all the data back to the client and request that the form be resubmitted with the correct information.

JavaScript provides a way to validate the form's data on the client's computer before sending it to the web server.

Example of form validation:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <div>
```

```
        <h1>Form event</h1>
        <form action="" method="post" id="myForm">
            <h6 class="output"></h6>
            <label for="Username">FirstName</label>
            <input type="text" name="firstname" id="firstname"
placeholder="Enter
firstname"> <br><br>
            <label for="Username">LastName</label>
            <input type="text" name="lastname" id="lastname"
placeholder="Enter
lastname"> <br><br>
            <label for="Username">Email</label>
            <input type="text" name="email" id="email"
placeholder="Enter email">
            <br><br>
            <label for="Username">Phone</label>
            <input type="text" name="phone" id="phone"
placeholder="Enter phone">
            <br><br>
            <label for="Password">Password</label>
            <input type="password" name="password" id="password"
placeholder="Enter
password"><br><br>
            <button type="submit" id="btn">Submit</button>
        </form>
    </div>

    <script src="hello.js"></script>
</body>

</html>
```

```
const myForm = document.getElementById("myForm");
const firstname = document.getElementById("firstname");
const lastname = document.getElementById("lastname");
const email = document.getElementById("email");
const phone = document.getElementById("phone");
const password = document.getElementById("password");
const output = document.querySelector(".output");
myForm.addEventListener("submit", formHandler);
```

```
function formHandler(e) {
  e.preventDefault();
  if (firstname.value.length === 0) {
    output.innerHTML = "Please enter your firstname";
    output.style.color = "red";
    output.style.fontSize = "1.9rem";
    output.style.fontFamily = "roboto";
  } else if (lastname.value.length === 0) {
    output.innerHTML = "Please enter your lastname";
    output.style.color = "red";
    output.style.fontSize = "1.9rem";
    output.style.fontFamily = "roboto";
  } else if (email.value.length === 0) {
    output.innerHTML = "Please enter your email";
    output.style.color = "red";
    output.style.fontSize = "1.9rem";
    output.style.fontFamily = "roboto";
  } else if (!isNaN(phone)) {
    output.innerHTML = "Please enter a valid phone number";
    output.style.color = "red";
    output.style.fontSize = "1.9rem";
    output.style.fontFamily = "roboto";
    console.log(typeof phone);
  } else if (password.value.length === 0) {
    output.innerHTML = "Please enter your password";
    output.style.color = "red";
    output.style.fontSize = "1.9rem";
    output.style.fontFamily = "roboto";
  } else if (password.value.length > 6) {
    output.innerHTML = "Password cannot exceed 6 characters";
    output.style.color = "red";
    output.style.fontSize = "1.9rem";
    output.style.fontFamily = "roboto";
  } else {
    output.innerHTML = "Congratulations! Forms successfully
submitted";
    output.style.color = "green";
    output.style.fontSize = "1.9rem";
    output.style.fontFamily = "roboto";
  }
}
```

CHAPTER 21

Asynchronous JavaScript

The word asynchronous means not occurring at the same time. What does it mean in the context of JavaScript?

Typically, executing things in sequence works well. But you may sometimes need to fetch data from the server or execute a function with a delay, something you do not anticipate occurring NOW. So, you want the code to execute asynchronously.

Asynchronous governs how we perform tasks which take some time to complete. Asynchronous start something and finish later. In order word asynchronous don't need to finish a task before executing another task.

HTTP Request

We make HTTP request to get data from another server. For example, when you type a web address you are performing a request. E.g., when you type https://www.facebook.com you are searching for that is responsible for such data. Then the server sends the HTTP response message.

We make this request to API end point. The API send back data in JSON format.

JSON

JSON stands for JavaScript Object Notation. JSON is a string that looks like JavaScript object. JSON is the format we get back once we send a HTTP request to a remote server.

Example of JSON

```
{"employees":[
    {"name":"Shyam", "email":"shyamjaiswal@gmail.com"},
    {"name":"Bob", "email":"bob32@gmail.com"},
    {"name":"Jai", "email":"jai87@gmail.com"}
]}
```

```
{
    "employee": {
        "name":       "sonoo",
        "salary":      56000,
        "married":    true
    }
}
```

API

API stands for Application Programming Interface. An API is simply a medium to fetch or send data between interfaces. Let's say you want to make an application that provides the user with some real-time data fetched from the server or maybe even allows you to modify or add data to some other endpoint. This is made possible by the API or the Application Programming Interface.

We will use a simple public API that requires no authentication and allows you to fetch some data by querying the API with GET requests. https://randomuser.me/ is a website that provides dummy data for random users that we can easily work with. We can get the response by making a request to https://randomuser.me/api/. The JSON response that we receive in the following format.

```
{
"results": [
{
"gender": "female",
"name": {
"title": "Miss",
"first": "Nina",
"last": "Simmmons"
```

```
},
"location": {
"street": {
"number": 970,
"name": "Eason Rd"
},
"city": "Fullerton",
"state": "Wyoming",
"country": "United States",
"postcode": 57089,
"coordinates": {
"latitude": "83.1807",
"longitude": "104.7170"
},
"timezone": {
"offset": "+8:00",
"description":
"Beijing, Perth, Singapore, Hong Kong"
}
},
"email": "nina.simmmons@example.com",
"login": {
"uuid": "bd0d135f-84df-4102-aa4f-5baaa41baf5c",
"username": "yellowfrog722",
"password": "dawg",
"salt": "q28gdiyN",
"md5": "291987daea22bb91775226574925b271",
"sha1": "a0463a26ea5c2ff4f3ad498fd01c5994926e5021",
"sha256":
"6583eb74ca08bfac50b3b29aa52c9f02ea5d9d017fef0e5a5a6fae4f5225f928
"
},
"dob": {
"date": "1980-11-01T23:10:05.403Z",
"age": 40
},
"registered": {
"date": "2013-04-02T02:26:52.904Z",
"age": 7
},
"phone": "(216)-693-7015",
"cell": "(501)-534-9413",
```

```
"id": {
"name": "SSN",
"value": "847-09-2973"
},
"picture": {
"large":
"https://randomuser.me/api/portraits/women/60.jpg",
"medium":
"https://randomuser.me/api/portraits/med/women/60.jpg",
"thumbnail":
"https://randomuser.me/api/portraits/thumb/women/60.jpg"
},
"nat": "US"
}
],
"info": {
"seed": "82a8d8d4a996ba17",
"results": 1,
"page": 1,
"version": "1.3"
}
```