What is JavaScript?

JavaScript is a powerful programming language that can add interactivity to a website. It was invented by Brendan Eich in 1997.

JavaScript is a lightweight, cross-platform, and interpreted compiled programming language which is also known as the scripting language for web pages. It is well-known for the development of web pages, many non-browser environments also use it. JavaScript can be used for **Client-side** developments as well as **Server-side** developments. Javascript is both an imperative and declarative type of language.

JavaScript contains a standard library of objects, like **Array**, **Date**, and **Math**, and a core set of language elements like **operators**, **control structures**, and **statements**.

console.log():-

The **console.log()** is a function in javascript that is used to print any kind of variable defined before in it or just print any message that needs to be displayed to the users.

Example :-

```
console.log("hello world….");

var a = 10;
console.log(a);

var b = "Hello";
console.log(b);

var ch = '2';
console.log(ch);

var a = 10;
console.log("A is: "+ a);

var b = "world";
console.log("Hello "+ b);
```

Variables:-

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

Ways to Declare a JavaScript Variable:

- Using `var`
- Using `let`
- Using `const`
- Using nothing

Example:-

```
var x = 5;
let b = 10;
const a = 3;
```

Always declare JavaScript variables with `var`,`let`, or `const`.

The `var` keyword is used in all JavaScript code from 1995 to 2015.

The `let` and `const` keywords were added to JavaScript in 2015.

If you want your code to run in older browsers, you must use `var`.

Identifiers :-

All JavaScript variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

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

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

Let:-

The `let` keyword was introduced in **ES6 (2015)**.

Variables defined with `let` cannot be **Redeclared**.

Variables defined with `let` must be Declared before use.

Variables defined with `let` have Block Scope.

Example:-

```
let x = "John Doe";

let x = 0;

// SyntaxError: 'x' has already been declared
```

Block Scope:-

Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope.**

ES6 introduced two important new JavaScript keywords: `let` and `const`.

These two keywords provide **Block Scope** in JavaScript.

Variables declared inside a { } block cannot be accessed from outside the block:

Example:-

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

Variables declared with the `var` keyword can NOT have block scope.

Variables declared inside a { } block can be accessed from outside the block.

Example:-

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

Redeclaring Variables:-

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

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

Example:-

```
var x = 10;

// Here x is 10

{

      var x = 2;       // Here x is 2

}

// Here x is 2
```

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

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

Example:-

```
let x = 10;

// Here x is 10

{

      let x = 2;       // Here x is 2

}

// Here x is 10
```

Const:-

The const keyword was introduced in ES6 (2015).

Variables defined with const cannot be **Redeclared**.

Variables defined with const cannot be **Reassigned**.

Variables defined with const have Block Scope.

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

When to use JavaScript const?

Always declare a variable with const when you know that the value should not be changed.

Use const when you declare:

- A new Array
- A new Object
- A new Function
- A new RegExp

Example:-

```
const PI = 3.141592653589793;

PI = 3.14;      // This will give an error

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

Block Scope:-

Declaring a variable with const is similar to let when it comes to Block Scope.

The x declared in the block, in this example, is not the same as the x declared outside the block :

Example:-

```
const x = 10;

// Here x is 10

{

      const x = 2;           // Here x is 2

}

// Here x is 10
```

Operators and Operands

The numbers or variables (in an arithmetic operation) are called operands.

The operation (to be performed between the two operands) is defined by an operator.

Operators:-

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Conditional Operators
- Type Operators

Arithmetic Operators:-

A typical arithmetic operation operates on two numbers.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (Division Remainder)
**	Exponentiation (ES2016)
++	Increment
--	Decrement

Example:-

```
let x = 10, y = 20;

console.log(x + y);    // 30

console.log(x - y);    // -10
```

```
console.log(x * y);    // 200
console.log(y / x);    // 2
console.log(y % x);    // 0
console.log(5 ** 2);   // 25
console.log(a++);      // 10
console.log(++a);      // 11
console.log(a--);      // 10
console.log(--a);      // 9
```

Assignment Operators:-

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

Example:-

```
let x = 10, y = 20;
let z = x;             // z = 10
console.log(x += y);   // 30
console.log(x -= y);   // -10
console.log(x *= y);   // 200
console.log(y /= x);   // 2
console.log(y %= x);   // 0
console.log(5 **= 2);  // 25
```

Comparison Operators:-

Operator	Description
==	Equal to
===	Equal value and equal type
!=	Not equal
!==	Not equal value and not equal type
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
? :	Ternary Operator

Example:-

```
console.log(10 == 10);       // true
console.log(10 === 10);      // true
console.log("0" == false);   // true
console.log(false == "0");   // true
console.log(false === "0");  // false
console.log(10 != 20);       // true
console.log(0 != false);     // true
console.log(10 > 20);        // false
console.log(10 >= 20);       // false
console.log(10 < 20);        // true
console.log(10 <= 20); // true
```

Type Operators:-

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Logical Operators:-

Operator	Description
&&	Logical And
||	Logical OR
!	Logical Not
^	Logical XOR

Example:-

```
let a = 2, b = 3, c;

c = --a && b++;  // if first condition value is 1 after it is continue

console.log(a,b,c);    // 1, 4, 3

c = --a || b++;        // 1, 3, 1

console.log(a != b);   // true

console.log(1 ^ 1);    // 0

console.log(0 ^ 1);    // 1

console.log(1 ^ 0);    // 1

console.log(0 ^ 0);    // 0

console.log(1^0^1);    // 0

console.log(0^1^0);    // 1

console.log(1^1^1);    // 1

console.log(0^0^0);    // 0
```

Operator Precedence:-

Multiplication (*) and division (/) have higher **precedence** than addition (+) and subtraction (-).

And (as in school mathematics) the precedence can be changed by using parentheses.

When using parentheses, the operations inside the parentheses are computed first.

Null(Object):-

The null value represents the intent absence of any object value. It is one of javascript's primitive values and is treated as false for boolean operations.

Undefined:-

A variable that has not been assigned a value is of type undefined .

Example:-

```
let a;

console.log(a);  // value is undefined, type  is undefined
```

Empty Values:-

An empty value has nothing to do with undefined. An empty string has a legal value and a type.

NaN:- (Not a number)

NaN type of number. Nan is a number that is not a legal number.

Example:-

```
let a;

a = 10*"s" ;

console.log(a);  // value is NaN, type  is Number


console.log(null == NaN);    // false

console.log(null === NaN);   // false


console.log(null == undefined);    // true

console.log(null === undefined);  // false


console.log(undefined == NaN);     // false

console.log(undefined === NaN);    // false
```

```
console.log(true == undefined);   // false
console.log(false == undefined);  // false

console.log(true == NaN);    // false
console.log(false == NaN);   // false

console.log(true == null);   // false
console.log(false == null);  // false
```

Conditional Statement:-

Conditional statements are used to perform different actions based on different conditions.

In JavaScript we have the following conditional statements:

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

If...else Statement:-

The `if` statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax:-

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

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

Syntax:-

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

The `else if` statement to specify a new condition if the first condition is false.

Syntax:-

```
if (condition1)
{
      //  block of code to be executed if condition1 is true
```

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

Switch Statement:-

The switch statement is used to perform different actions based on different conditions.

Syntax:-

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

This is how it works:

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

Switching Details:-

If multiple cases matches a case value, the first case is selected.

If no matching cases are found, the program continues to the default label.

If no default label is found, the program continues to the statement(s) after the switch.

Loop Statement:-

Loops can execute a block of code a number of times.

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

For Loop :-

The `for` statement creates a loop with 3 optional expressions.

Syntax:-

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

Expression 1 is executed (one time) before the execution of the code block.

Expression 2 defines the condition for executing the code block.

Expression 3 is executed (every time) after the code block has been executed.

Example:-

```
let text = "";

for (let i = 1; i <= 5; i++) {

  text = text + i + " ";

} // 1  2  3  4  5
```

For in Loop :-

The JavaScript `for in` statement loops through the properties of an Object.

Syntax:-

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

Example:-

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

```
let text = "";

for (let x in person) {

  text += person[x];

} // John Doe 25
```

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

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

let text = "";

for (let x in number) {

  text += number[x];

} // 45 4 9 16 25
```

For of Loop :-

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

Syntax:-

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

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

Example:-

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

let text = "";

for (let x of cars) {

  text += x + “\t”;

} // BMW   Volvo   Mini


let language = "JavaScript";
```

```
let text = "";

for (let x of language ) {

  text += x + “ ”; } // J a v a S c r i p t
```

While Loop :-

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

Syntax:-

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

Example:-

```
while (i < 10) {

  text += i + "\t";

  i++;

} // 0 1 2 3 4 5 6 7 8 9
```

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 {

  text += i + "\t";

  i++;

} while (i < 10); // 0 1 2 3 4 5 6 7 8 9
```

Break Statement :-

The break statement "jumps out" of a loop.

Example:-

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += i + "\t";
} // 0 1 2
```

Continue Statement :-

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

Example:-

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { continue; }
  text += i + "\t";
} // 0 1 2 4 5 6 7 8 9
```

Map :-

A Map holds key-value pairs where the keys can be any datatype.
A Map remembers the original insertion order of the keys.

Method	Description
new Map()	Creates a new map
set()	Sets the value for a key in a Map
get()	Gets the value for a key in a Map
delete()	Remove a Map Element specified by the key
has()	Returns a true if a key exists in a map
forEach()	Calls a function for each key/value pair in a Map
entries()	Returns an iterator with the [key, value] pairs in a Map
size	Returns a number of elements in a Map

Example:-

```
// Create a Map
 const fruits = new Map([["apples", 500],["bananas", 300],
                         ["oranges", 200]]);
// Set Map Values
const fruits = new Map();
fruits.set("apples", 500);
fruits.set("bananas", 300);
// Get Map Values
fruits.get("apples");    // Returns 500
```

The `size` property returns the number of elements in a Map.

```
fruits.size;     // 3
```

Set :-

A JavaScript Set is a collection of unique values.
Each value can only occur once in a Set.

Method	Description
new Set()	Creates a new Set
add()	Adds a new element to the Set
delete()	Remove an element from a Set
has()	Returns true if a value exists in the Set
clear()	Removes all elements from a Set
forEach()	Invokes a callback for each element in the Set
entries()	Returns an iterator with all the values in a Set
size	Returns a number of elements in a Set

Example:-

```
 const letters = new Set(["a","b","c"]);      // Create a Set
letters.add("a");      // Add Values to the Set
letters.add("b");
letters.add("c");
```

Shallow Copy:

A shallow copy is a copy that only goes one level deep. In other words, it copies the object and all its properties, but any nested objects or arrays will still reference the same memory location as the original object. It means that if you make changes to the nested object, it will also affect the original object, as well as the copied object.

Example:-

```
 const first_person = {
           name: "Jack",
           age: 24,
     }
const second_person = first_person;

second_person.age = 25;

console.log(first_person.age); // output: 25
console.log(second_person.age); // output: 25
```

Deep Copy:

A deep copy is a copy that creates a new object with new memory locations for all of its properties and nested objects or arrays. It means that if you make changes to the copied object or any of its nested objects or arrays, it will not affect the original object.

Example:-

```
 const first_person = {
           name: "Jack",
           age: 24,
     }
const second_person = { ...first_person};

second_person.age = 25;

console.log(first_person.age); // output: 24
console.log(second_person.age); // output: 25
```

String

A JavaScript string is zero or more characters written inside quotes.

You can use single or double quotes.

Example:-

```
let a = “Hello World”; // Double quotes

let a = ‘Hello World’; // Single quotes
```

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

Example:-

```
let answer = “It’s Beautiful…”;

let answer = “He is called ‘Johnny’”;

let answer = ‘He is called “Johnny”’;
```

String Methods

Types of String methods are given below:

- **String length**
- **String slice()**
- **String substring()**
- **String substr()**
- **String replace()**
- **String replaceAll()**
- **String toUpperCase()**
- **String toLowerCase()**
- **String concat()**
- **String trim()**
- **String trimStart()**
- **String trimEnd()**
- **String padStart()**
- **String padEnd()**
- **String charAt()**
- **String charCodeAt()**
- **String split()**

String Length:-

The `length` property returns the length of a string.

Example:-

```
let text = “Hello World”;

let length = text.length;    // length = 11
```

String Slice():-

slice() 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 length = text.slice(7, 13);    // Banana
```

Notes:- **JavaScript counts positions from zero.**

1. If you omit the second parameter, the method will slice out the rest of the string.

Example:-

```
let text = “Apple, Banana, Kiwi”;

let length = text.slice(7);  // Banana, Kiwi
```

2. If a parameter is negative, the position is counted from the end of the string.

Example:-

```
let text = “Apple, Banana, Kiwi”;

let length = text.slice(-12);      // Banana, Kiwi


let text = “Apple, Banana, Kiwi”;

let length = text.slice(-12,-6);   // Banana
```

String substring():-

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

Example:-

```
let text = “Apple, Banana, Kiwi”;

let length = text.substring(7, 13);      // Banana
```

String substr():-

substr() is similar to slice().
The difference is that the second parameter specifies the **length** of the extracted part.

Example:-

```
let text = “Apple, Banana, Kiwi”;

let length = text.substr(7, 6);    // Banana
```

1. If you omit the second parameter, substr() will slice out the rest of the string.

Example:-

```
let text = “Apple, Banana, Kiwi”;

let length = text.substr(7); // Banana, Kiwi
```

2. If the first parameter is negative, the position is counted from the end of the string.

Example:-

```
let text = “Apple, Banana, Kiwi”;

let length = text.substr(-4);      // Kiwi
```

String replace():-

The replace() method replaces a specified value with another value in a string

Example:-

```
let text = “Please visit Microsoft!”;

let length = text.replace("Microsoft", "SkillQode");
```

Notes:- The replace() method does not change the string it is called on.

The replace() method returns a new string.

The replace() method replaces only the first match

If you want to replace all matches, use a regular expression with the /g flag set.

The replace() method is case sensitive.

String replaceAll():-

The `replaceAll()` method allows you to specify a regular expression instead of a string to be replaced.

Example:-

```
let text = "I love cats. Cats are very easy to love. Cats are very popular.";

text = text.replaceAll("Cats","Dogs");

text = text.replaceAll("cats","dogs");

// I love dogs. Dogs are very easy to love. Dogs are very popular.
```

String toUpperCase():-

The `toUpperCase()` method converts string lower case to upper case.

Example:-

```
let text1 = "Hello World!";

let text2 = text1.toUpperCase();  // HELLO WORLD!
```

String toLowerCase():-

The `toUpperCase()` method converts string upper case to lower case.

Example:-

```
let text1 = "Hello World!";

let text2 = text1.toLowerCase();  // hello world!
```

String concat():-

The `concat()` joins two strings.

Example:-

```
let text1 = "Hello";

let text2 = "World";

let text3 = text1.concat(“ ”, text2);   // Hello World
```

The `concat()` method can be used instead of the plus operator.

String trim():-

The `trim()` method removes whitespace from both sides of a string.

Example:-

```
let text1 = "       Hello World!      ";
let text2 = text1.trim();    // Hello world!
```

String trimStart():-

The `trimStart()` method removes like `trim()`, but removes whitespace only from the start of a string.

Example:-

```
let text1 = "       Hello World!     ";
let text2 = text1.trim();    // Hello world!
```

String trimEnd():-

The `trimEnd()` method removes like `trim()`, but removes whitespace only from the end of a string.

Example:-

```
let text1 = "       Hello World!     ";
let text2 = text1.trim();    //  Hello world!
```

String padStart():-

The `padStart()` method pads a string with another string.

Example:-

```
let text = "5";
let padded = text.padStart(4,"0");      //  0005
```

To pad a number, convert the number to a string first.

Example:-

```
let num = 5;
```

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

let padded = text.padStart(4,0);        //  0005
```

String padEnd():-

The padEnd() method pads a string with another string.

Example:-

```
let text = "5";

let padded = text.padEnd(4,"0");        //  5000
```

To pad a number, convert the number to a string first.

Example:-

```
let num = 5;

let text = num.toString();

let padded = text.padEnd(4,0);          //  5000
```

String 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);        //  H
```

String charCodeAt():-

The charCodeAt() method returns the unicode of the character at a specified index in a string.
The method returns a UTF-16 code (an integer between 0 and 65535).

Example:-

```
let text = "HELLO WORLD";

let char = text.charCodeAt(0);          // 72
```

String split():-

A string can be converted to an array with the `split()` method.

Example:-

```
let text = "HELLO WORLD";

let char = text.split(",")    // Split on commas

let char = text.split(" ")    // Split on spaces

let char = text.split("|")    // Split on pipe
```

Maths Object

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

Example:-

```
Math.PI;
```

The Math object is static.

All methods and properties can be used without creating a Math object first.

Math Properties(Constants):-

The syntax for any Math property is : `Math.property`
JavaScript provides 8 mathematical constants that can be accessed as Math properties.

Example:-

```
Math.E        // returns Euler's number
Math.PI       // returns PI
Math.SQRT2    // returns the square root of 2
Math.SQRT1_2  // returns the square root of 1/2
Math.LN2      // returns the natural logarithm of 2
Math.LN10     // returns the natural logarithm of 10
Math.LOG2E    // returns base 2 logarithm of E
Math.LOG10E   // returns base 10 logarithm of E
```

Math Methods

The syntax for Math any methods is : `Math.method(number)`

Math.round():-

`Math.round(x)` returns the nearest integer.

Example:-

```
Math.round(4.6);       // 5
Math.round(4.3);       // 4
```

Math.ceil():-

`Math.ceil(x)` returns the value of x rounded **up** to its nearest integer.

Example:-

```
Math.ceil(4.6);       // 5
Math.ceil(4.2);       // 5
Math.ceil(-4.3);      // -4
```

Math.floor():-

`Math.floor(x)` returns the value of x rounded **down** to its nearest integer.

Example:-

```
Math.floor(4.6);      // 4
Math.floor(4.2);      // 4
Math.floor(-4.3);     // -5
```

Math.trunc():-

`Math.trunc(x)` returns the integer part of x.

Example:-

```
Math.trunc(4.6);      // 4
Math.trunc(4.2);      // 4
```

Math.sign():-

`Math.sign(x)` returns if x is negative, null or positive.

Example:-

```
Math.sign(-4);        // -1
Math.sign(0);         // 0
Math.sign(4);         // 1
```

Math.pow():-

`Math.pow(x, y)` returns the value of x to the power of y.

Example:-

```
Math.pow(2, 3);        // 8
Math.pow(8, 2);        // 64
```

Math.sqrt():-

`Math.sqrt(x)` returns the square root of x.

Example:-

```
Math.sqrt(16);         // 4
Math.sqrt(121);        // 11
```

Math.abs():-

`Math.abs(x)` returns the absolute (positive) value of x.

Example:-

```
Math.abs(-6);          // 6
Math.abs(1.2);         // 1.2
```

Math.min() and Math.max():-

`Math.min()` and `Math.max()` can be used to find the lowest or highest value in a list of arguments.

Example:-

```
Math.min(0, 150, 30, 20, -8, -200);     // -200
Math.max(0, 150, 30, 20, -8, -200);     // 150
```

Math.random():-

`Math.random()` returns a random number between 0 (inclusive), and 1 (exclusive).

Example:-

```
Math.random();         // 0.3860669187533996
```

Math.sin():-

Math.sin(x) returns the sine (a value between -1 and 1) of the angle x (given in radians).
If you want to use degrees instead of radians, you have to convert degrees to radians:

Angle in radians = Angle in degrees x PI / 180.

Example:-

```
Math.sin(90 * Math.PI / 180); //  1 (the sine of 90 degrees)

Math.sin(30 * Math.PI / 180); //  0.5 (the sine of 30 degrees)
```

Math.cos():-

Math.cos(x) returns the cosine (a value between -1 and 1) of the angle x (given in radians).
If you want to use degrees instead of radians, you have to convert degrees to radians:

Angle in radians = Angle in degrees x PI / 180.

Example:-

```
Math.cos(0 * Math.PI / 180); //  1 (the cos of 0 degrees)

Math.cos(60 * Math.PI / 180); //  0.5 (the cos of 60 degrees)
```

Math.log():-

Math.log(x) returns the natural logarithm of x.
The natural logarithm returns the time needed to reach a certain level of growth.

Example:-

```
Math.log(1);     //  0

Math.log(2);     //  0.6931471805599453

Math.log(10);    //  2.302585092994046
```

Math.log2():-

Math.log2(x) returns the base 2 logarithm of x.

Example:-

```
Math.log2(8);    //  3
```

Math.log10():-

`Math.log10(x)` returns the base 10 logarithm of x.

Example:-

```
Math.log10(1000);//  3
```

Number Method

These **number methods** can be used on all JavaScript numbers:

Method	Description
toString()	Returns a number as a string
toExponential()	Returns a number written in exponential notation
toFixed()	Returns a number written with a number of decimals
toPrecision()	Returns a number written with a specified length
Valueof()	Returns a number as a number

Example:-

```
let x = 123;
x.toString();    //  123
(123).toString();//  123
(100 + 23).toString(); //  123

let x = 9.656;

x.toExponential();     //  9.656e+0
x.toExponential(2);    //  9.66e+0
x.toExponential(4);    //  9.6560e+0
x.toExponential(6);    //  9.656000e+0

let x = 9.656;
x.toFixed();     //  10
x.toFixed(2);    //  9.66
x.toFixed(4);    //  9.6560
x.toFixed(6);    //  9.656000

let x = 9.656;
x.toPrecision(); //  9.656
x.toPrecision(2);//  9.7
x.toPrecision(4);//  9.656
x.toPrecision(6);//  9.65600

let x = 123;
x.ValueOf();     //  123
(123).ValueOf(); //  123
(100 + 23).ValueOf();  //  123
```

In JavaScript, a number can be a primitive value (typeof = number) or an object (typeof = object).

The `valueOf()` method is used internally in JavaScript to convert Number objects to primitive values.

Converting Variable to Numbers

There are 3 JavaScript methods that can be used to convert a variable to a number:

Method	Description
Number()	Returns a number converted from argument.
parseFloat()	Parse its argument and returns a floating point number
parseInt()	Parse its argument and returns a whole number

Example:-

```
Number(true);    //  1
Number(false);   //  0
Number("10");    //  10
Number("  10");  //  10
Number("10  ");  //  10
Number(" 10  "); //  10
Number("10.33"); //  10.33
Number("10,33"); //  NaN
Number("10 33"); //  NaN
Number("John");  //  NaN

parseInt("-10");       //  -10
parseInt("-10.33");    //  -10
parseInt("10");        //  10
parseInt("10.33");     //  10
parseInt("10 20 30");  //  10
parseInt("10 years");  //  10
parseInt("years 10");  //  NaN

parseFloat("10");      //  10
parseFloat("10.33");   //  10.33
parseFloat("10 20 30");      //  10
parseFloat("10 years");      //  10
parseFloat("years 10");      //  NaN
```

Number Object Method

These **object methods** belong to the **Number** object:

Method	Description
Number.isInteger()	Returns true if the argument is an integer
Number.isSafeInteger()	Returns true if the argument is a safe integer
Number.parseFlaot()	Convert a string to a number
Number.parseInt()	Convert a string to a whole number

Example:-

```
Number.isInteger(10);  //  true
Number.isInteger(10.5);      //  false

Number.isSafeInteger(10);    //  true
Number.isSafeInteger(12345678901234567890);   //  false

Number.parseFloat("10");           //  10
Number.parseFloat("10.33");  //  10.33
Number.parseFloat("10 20 30");     //  10
Number.parseFloat("10 years");     //  10
Number.parseFloat("years 10");     //  NaN

Number.parseInt("-10");            //  -10
Number.parseInt("-10.33");         //  -10
Number.parseInt("10");             //  10
Number.parseInt("10.33");          //  10
Number.parseInt("10 6");           //  10
Number.parseInt("10 years");       //  10
Number.parseInt("years 10");       //  NaN
```

Date Object

Date objects are static. The "clock" is not "running".

The computer clock is ticking, date objects are not.

By default, JavaScript will use the browser's time zone and display a date as a full text string:

Wed Dec 28 2022 17:44:57 GMT+0530 (India Standard Time)

Creating Date Object

Date objects are created with the `new Date()` constructor.

There are **9 ways** to create a new date object

```
new Date()

new Date(date string)

new Date(year,month)

new Date(year,month,day)

new Date(year,month,day,hours)

new Date(year,month,day,hours,minutes)

new Date(year,month,day,hours,minutes,seconds)

new Date(year,month,day,hours,minutes,seconds,ms)

new Date(milliseconds)
```

Notes:-

JavaScript counts months from **0** to **11**. **January = 0** and **December = 11**

Specifying a month higher than 11, will not result in an error but add the overflow to the next year.

Specifying a day higher than max, will not result in an error but add the overflow to the next month.

One and two digit years will be interpreted as 19xx.

Example:-

```
const d = new Date();
// Wed Dec 28 2022 17:50:34 GMT+0530 (India Standard Time)

const d = new Date("October 13, 2014 11:13:00"); // (Date String)
// Mon Oct 13 2014 11:13:00 GMT+0530 (India Standard Time)

const d = new Date(2018, 11, 24, 10, 33, 30, 0); // (year,month,....)
// Mon Dec 24 2018 10:33:30 GMT+0530 (India Standard Time)
```

```
const d = new Date(99, 11, 24); // Previous Century
// Fri Dec 24 1999 00:00:00 GMT+0530 (India Standard Time)


const d = new Date(100000000000); // (Milliseconds)
// Sat Mar 03 1973 15:16:40 GMT+0530 (India Standard Time)
```

Date Methods

When a date object is created, a number of **methods** allow you to operate on it.

Date methods allow you to **get** and **set** the year, month, day, hour, minute, second, and millisecond of date objects, using either **local time** or **UTC** (universal, or GMT) time.

When you display a date object in HTML, it is automatically converted to a string, with the `toString()` method.

Example:-

```
const d = new Date();
d.toString();
// Wed Dec 28 2022 17:50:34 GMT+0530 (India Standard Time)
```

The `toDateString()` method converts a date to a more readable format.

Example:-

```
const d = new Date();
d.toDateString();
// Wed Dec 28 2022
```

The `toUTCString()` method converts a date to a string using the UTC standard.

Example:-

```
const d = new Date();
d.toUTCString();
// Wed Dec 28 2022 17:50:34 GMT
```

The `toISOString()` method converts a date to a string using the ISOstandard. ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ)

Example:-

```
const d = new Date();
d.toISOString();
// 2022-12-28T05:05:50.697Z
```

Date Get Methods

Method	Description
getFullYear()	Get **Year** as a four digit number (yyyy)
getMonth()	Get **Month** as a number (0-11)
getDate()	Get **Day** as a number (1-31)
getDay()	Get **WeekDay** as a number (0-6)
getHours()	Get **Hour** (0-23)
getMinutes()	Get **Minutes** (0-59)
getSeconds()	Get **Seconds** (0-59)
getMilliSeconds()	Get **MilliSeconds** (0-999)
getTime()	Get **Time** (milliseconds since january 1, 1970)

UTC methods use UTC time (Coordinated Universal Time).

UTC time is the same as GMT (Greenwich Mean Time).

The difference between Local time and UTC time can be up to 24 hours.

The `getTimezoneOffset()` method returns the difference (in minutes) between local time in UTC time.

Example:-

```
let diff = d.getTimezoneOffset();       // -330
```

Date Set Methods

Method	Description
setFullYear()	Set the **Year** (optionally month and day)
setMonth()	Set the **Month** as a number (0-11)
setDate()	Set the **Day** as a number (1-31)
setDay()	Set the **WeekDay** as a number (0-6)
setHours()	Set the **Hours** (0-23)
setMinutes()	Set the **Minutes** (0-59)
setSeconds()	Set the **Seconds** (0-59)
setMilliSeconds()	Set the **MilliSeconds** (0-999)
setTime()	Set the **Time** (milliseconds since january 1, 1970)

Array

An array is a special variable, which can hold more than one value.
An array can hold many values under a single name, and you can access the values by referring to an index number.

Syntax:-

```
const array_name = [item1, item2, ...];
```

Example:-

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

const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";

const cars = new Array("Saab", "Volvo", "BMW");
```

Array length:-

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

Example:-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.length;       // 4
```

Array Push:-

The easiest way to add a new element to an array is using the `push()` method.
The `push()` method returns the new array length.

Example:-

```
const fruits = ["Banana", "Orange", "Apple"];
fruits.push("Lemon");  // Adds a new element (Lemon) to fruits
```

New element can also be added to an array using the `length` property:

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[fruits.length] = "Lemon";  // Adds "Lemon" to fruits
```

Array Pop:-

The `pop()` method removes the **last element** from an **array**.
The `pop()` method returns the value that was **"popped out"**.

Example:-

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

Array Shift:-

The `shift()` method **removes** the **first array element** and **"shifts"** all other **elements to a lower index**.
The `shift()` method returns the value that was **"shifted out"**.

Example:-

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

Array Unshift:-

The `unshift()` method **adds a new element** to an array (**at the beginning**), and "unshifts" older elements.
The `unshift()` method returns the new array length.

Example:-

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

Array Changing Elements:-

Array elements are accessed using their **index number**.

Example:-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
Fruits[0] = "Kiwi";    // Kiwi,Orange,Apple,Mango
```

Array Merging(Concatenating):-

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);
      // Cecilie,Lone,Emil,Tobias,Linus
```

The `concat()` method can take any number of array arguments.

Example:-

```
const arr1 = ["Cecilie", "Lone"];
const arr2 = ["Emil", "Tobias", "Linus"];
const arr3 = ["Robin", "Morgan"];
const myChildren = arr1.concat(arr2, arr3);
      // Cecilie,Lone,Emil,Tobias,Linus,Robin,Morgan
```

The `concat()` method can also take strings as arguments.

Example:-

```
const arr1 = ["Emil", "Tobias", "Linus"];
const myChildren = arr1.concat("Peter");
      // Emil,Tobias,Linus,Peter
```

Array Splice():-

The `splice()` method adds new items to an array.

Example:-

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

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**.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 2, "Lemon", "Kiwi");
      // Original Array:- Banana,Orange,Apple,Mango
         New Array:- Banana,Orange,Lemon,Kiwi
         Removed Items:- Apple,Mango
```

With clever parameter setting, you can use `splice()` to remove elements without leaving "holes" in the array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0, 1);   // Orange,Apple,Mango
```

The first parameter (0) defines the position where new elements should be **added** (spliced in). The second parameter (1) defines **how many** elements should be **removed**. The rest of the parameters are omitted.No new elements will be added.

Array Slice():-

The `slice()` method slices out a piece of an array into a new array.
The `slice()` method creates a new array.
The `slice()` method does not remove any elements from the source array.

Example:-

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

The method then selects elements from the start argument, and up to (but not including) the end argument.

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1, 3);
      // Orange,Lemon
```

Array Sort():-

The `sort()` method sorts an array alphabetically.

Example:-

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

Numeric Sort:-

default, the `sort()` function sorts values as strings.
This works well for strings ("Apple" comes before "Banana").
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 results when sorting numbers.
You can fix this by providing a **compare function.**

Example:-

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
      // 1,5,10,25,40,100
```

Array Reverse():-

The `reverse()` method reverses the elements in an array.

Example:-

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

Array Max():-

The `Math.max.apply` to find the highest number in an array.

Example:-

```
function myArrayMax(arr) {
      return Math.max.apply(null, arr);
}
const points = [40, 100, 1, 5, 25, 10];
myArrayMax(points);
      // The highest number is 100.
```

Array Min():-

The `Math.min.apply` to find the lowest number in an array.

Example:-

```
function myArrayMin(arr) {
      return Math.min.apply(null, arr);
}
const points = [40, 100, 1, -5, 25, 10];
myArrayMin(points);
      // The lowest number is -5.
```

Array forEach():-

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

Example:-

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);

function myFunction(value, index, array) {
      txt += value + " ";
}
      // 45,4,9,16,25
```

Array map():-

The `map()` method creates a new array by performing a function on each array element.
The `map()` method does not execute the function for array elements without values.
The `map()` method does not change the original array.

Example:-

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);

function myFunction(value, index, array) {
      return value * 2;
}
      // 90,8,18,32,50.
```

Array filter():-

The `filter()` method creates a new array with array elements that pass a test.

Example:-

```
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);

function myFunction(value, index, array) {
      return value > 18;
}
      // 45,25.
```

Array reduce():-

The `reduce()` method runs a function on each array element to produce (reduce it to) a single value.
The `reduce()` method works from left-to-right in the array. See also `reduceRight()`.

Example:-

```
const numbers = [45, 4, 9, 16, 25];
const sum = numbers.reduce(myFunction);

function myFunction(total, value, index, array) {
      return total + value;
}
      // 99.
```

Array reduceRight():-

The `reduceRight()` method runs a function on each array element to produce (reduce it to) a single value.
The `reduceRight()` works from right-to-left in the array.

Example:-

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduceRight(myFunction);

function myFunction(total, value, index, array) {
       return total + value;
}
      // 99.
```

Array every():-

The `every()` method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

Example:-

```
const numbers = [45, 4, 9, 16, 25];
let allOver18 = numbers.every(myFunction);

function myFunction(value, index, array) {
      return value > 18;
}
      // false.
```

Array some():-

The `some()` method tests whether at least one element in the array passes the test implemented by the provided function. It returns true if, in the array, it finds an element for which the provided function returns true; otherwise it returns false. It doesn't modify the array.

Example:-

```
const numbers = [45, 4, 9, 16, 25];
let someOver18 = numbers.some(myFunction);

function myFunction(value, index, array) {
       return value > 18;
}
      // true.
```

Array find():-

The `find()` method returns the first element in the provided array that satisfies the provided testing function.
If no values satisfy the testing function, **undefined** is returned.

Example:-

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);

function myFunction(value, index, array) {
      return value > 18;
}
      // 25
```

Array findIndex():-

The `findIndex()` method returns the index of the first element in an array that satisfies the provided testing function. If no elements satisfy the testing function, -1 is returned.

Example:-

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(myFunction);

function myFunction(value, index, array) {
      return value > 18;
}          // 3
```

Array entries():-

The `entries()` method returns a new **Array Iterator** object that contains the key/value pairs for each index in the array.

Example:-

```
const array1 = ['a', 'b', 'c'];
const iterator1 = array1.entries();
console.log(iterator1.next().value);      // [0, "a"]
console.log(iterator1.next().value);      // [1, "b"]
```

Array includes():-

The `Array.includes()` to arrays. This allows us to check if an element is present in an array (including NaN, unlike indexOf).
The `Array.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"];
fruits.includes("Mango"); // is true
```

Array copyWithin():-

The `copyWithin()` method copies array elements to another position in the array.
The `copyWithin()` method overwrites the existing values.
The `copyWithin()` method does not add items to the array.

Syntax:-

```
array.copyWithin(target, start, end)
```

Example:-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.copyWithin(2, 0));    // Banana,Orange,Banana,Orange

const fruits = ["Banana", "Orange", "Apple", "Mango", "Kiwi"];
console.log(fruits.copyWithin(2, 0, 2));
// Banana,Orange,Banana,Orange,Kiwi,Papaya
```

Function

JavaScript functions are defined with the `function` keyword.
You can use a function declaration or a function expression.

Function Declaration:-

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

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

Example:-

```
function myFunction(a, b) {
       return a * b;
}
myFunction(2,5)  // 10
```

Function Expression:-

A JavaScript function can also be defined using an expression.

Example:-

```
const x = function (a, b) {return a * b};
let z = x(4, 3); // 12
```

- The function above is actually an **anonymous function** (a function without a name).
- Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

Function Constructor:-

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

Example:-

```
const myFunction = new Function("a", "b", "return a * b");
let x = myFunction(4, 3);    // 12
```

You actually don't have to use the function constructor.

Function Hoisting:-

- Hoisting is JavaScript's default behaviour of moving declarations to the top of the current scope.
- Hoisting applies to variable declarations and to function declarations.

Example:-

```
myFunction(5);

function myFunction(y) {
      return y * y;
}           // 25
```

Functions defined using an expression are not hoisted.

Self-invoking Function:-

- Function expressions can be made "self-invoking".
- A self-invoking expression is invoked (started) automatically, without being called.
- Function expressions will execute automatically if the expression is followed by ().
- You cannot self-invoke a function declaration.
- You have to add parentheses around the function to indicate that it is a function expression.

Example:-

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

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

Function are Object:-

- The `typeof` operator in JavaScript returns "function" for functions.
- But, JavaScript functions can best be described as objects.
- JavaScript functions have both properties and methods.
- The `arguments.length` property returns the **number of arguments received** when the function was invoked.

Example:-

```
function myFunction(a, b) {
      return arguments.length; }
let text = myFunction(4,3);  // total number of arguments is 2
```

The `toString()` method returns the function as a string.

Example:-

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

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

Arrow Function:-

- Arrow functions allow a short syntax for writing function expressions.
- You don't need the `function` keyword, the `return` keyword, and the **curly brackets.**

Example:-

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

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

- Arrow functions do not have their own `this`. They are not well suited for defining **object methods**.
- Arrow functions are not hoisted. They must be defined **before** they are used.
- Using `const` is safer than using `var`, because a function expression is always constant value.
- You can only omit the `return` keyword and the curly brackets if the function is a **single statement**.

Function Parameters:-

- A JavaScript **`function`** does not perform any checking on parameter values (arguments).

Example:-

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

- Function **parameters** are the **names** listed in the function definition.
- Function **arguments** are the real **values** passed to (and received by) the function.

Function Rest Parameters:-

- The rest parameter (...) allows a function to treat an indefinite number of arguments as an array.

Example:-

```
function sum(...args) {
      let sum = 0;
      for (let arg of args) sum += arg;
      return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);   // 326
```

Arguments Object:-

- JavaScript functions have a built-in object called the arguments object.
- The argument object contains an array of the arguments used when the function was called (invoked).
- you can simply use a function to find (for instance) the highest value in a list of numbers.

Example:-

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
      let sum = 0;
      for (let i = 0; i < arguments.length; i++) {
            sum += arguments[i];
       }
      return sum;
}     // 871
```

Arguments are Passed By Value:-

- The parameters, in a function call, are the function's arguments.
- JavaScript arguments are passed by **value**: The function only gets to know the values, not the argument's locations.
- If a function changes an argument's value, it does not change the parameter's original value.

- **Changes to arguments are not visible (reflected) outside the function.**

Object are Passed By Reference:-

- In JavaScript, object references are values.
- Because of this, objects will behave like they are passed by **reference**.
- If a function changes an object property, it changes the original value.
- **Changes to object properties are visible (reflected) outside the function.**

Generator Function:-

- A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return.
- The yield statement suspends the function's execution and sends a value back to the caller, but retains enough state to enable the function to resume where it is left off. When resumed, the function continues execution immediately after the last yield.

Example:-

```
function* generate() {
       yield 1;
       yield 2;
       return 3;
}     // "generator function" creates "generator object"
let generator = generate();
console.log(generator.next().value); // 1
console.log(generator.next().value); // 2
console.log(generator.next().value); // 3
```

The primary method of a generator is `next()`. When you call it, it runs the execution till the nearest yield `<value>` statement.

this Keyword:-

Example:-

```
const person = {
      firstName: "John",
      lastName : "Doe",
      id    : 5566,
```

```
        fullName : function() {
              return this.firstName + " " + this.lastName;
              }
};            // John Doe
```

- The `this` keyword refers to an object.
- Which object depends on how `this` is being invoked (used or called).

- The `this` keyword refers to different objects depending on how it is used.
    - In an object method, `this` refers to the object.
    - Alone, `this` refers to the global object.
    - In a function, `this` refers to the global object.
    - In a function, in strict mode, `this` is `undefined`.
    - In an event, `this` refers to the element that received the event.
    - Methods like `call()`, `apply()`, and `bind()` can refer `this` to any object.

call() Method:-

- The `call()` method is a predefined JavaScript method.
- It can be used to invoke (call) a method with an owner object as an argument (parameter).
- With `call()`, an object can use a method belonging to another object.

Example:-

```
const person = {
      fullName: function() {
      return this.firstName + " " + this.lastName;
      }
}
const person1 = {
      firstName:"John",
      lastName: "Doe"
      }
const person2 = {
      firstName:"Mary",
      lastName: "Doe"
      }
person.fullName.call(person1);     // This will return "John Doe":
```

apply() Method:-

- The `apply()` method is similar to the `call()` method.
- With the `apply()` method, you can write a method that can be used on different objects.

Example:-

```
const person = {
      fullName: function() {
      return this.firstName + " " + this.lastName;
      }
}
const person1 = {
      firstName:"John",
      lastName: "Doe"
      }
person.fullName.apply(person1);   // This will return "John Doe":
```

The `call()` method takes arguments **separately**.
The `apply()` method takes arguments as an **array**.

Example:-

```
      const person = {
            fullName: function(city, country) {
            return this.firstName + " " + this.lastName + "," + city + "," +
country;    }
      }
      const person1 = {
            firstName:"John",
            lastName: "Doe"
            }
      person.fullName.apply(person1, ["Oslo", "Norway"]);
            // This will return "John Doe,Oslo,Norway"
```

bind() Method:-

- With the `bind()` method, an object can borrow a method from another object.

Example:-

```
const person = {
            firstName:"John",
            lastName: "Doe",
            fullName: function () {
                  return this.firstName + " " + this.lastName;
            }
}
const member = {
            firstName:"Hege",
            lastName: "Nilsen",
}
let fullName = person.fullName.bind(member);  // Hege Nilsen
```

Object

In JavaScript, almost "everything" is an object.

- Booleans can be objects (if defined with the `new` keyword)
- Numbers can be objects (if defined with the `new` keyword)
- Strings can be objects (if defined with the `new` keyword)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are always objects

All JavaScript values, except primitives, are objects.

Creating a javascript Object

There are different ways to create new objects:

- Create a single object, using an object literal.
- Create a single object, with the keyword `new`.
- Define an object constructor, and then create objects of the constructed type.
- Create an object using `Object.create()`.

Example:-

```
// using Object Literal
const person = {
      firstName: "John",
      lastName: "Doe",
      age: 50,
      eyeColor: "blue"
};

// using new Object keyword
const person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

Object Properties

Properties are the values associated with a JavaScript object.
A JavaScript object is a collection of unordered properties.
Properties can usually be changed, added, and deleted, but some are read only.

Syntax:-

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

Adding Properties:

You can add new properties to an existing object by simply giving it a value.

Example:

```
person.nationality = "Indian";
```

Deleting Properties:

The `delete` keyword deletes a property from an object:

Example:

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

The `delete` keyword deletes both the value of the property and the property itself.
The `delete` operator is designed to be used on object properties. It has no effect on variables or functions.

Callbacks

A callback is a function passed as an argument to another function.
This technique allows a function to call another function.
A callback function can run after another function has finished.

Example:-

```
function myDisplayer(some) {
      document.getElementById("demo").innerHTML = some;
      }

function myCalculator(num1, num2, myCallback) {
      let sum = num1 + num2;
      myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

Promises

"Producing code" is code that can take some time.
"Consuming code" is code that must wait for the result.
A Promise is a JavaScript object that links producing code and consuming code.

Example:

```
let myPromise = new Promise(function(myResolve, myReject) {
      // "Producing Code" (May take some time)

      myResolve(); // when successful
      myReject();  // when error
});

      // "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
      function(value) { /* code if successful */ },
      function(error) { /* code if some error */ }
);
```

When the producing code obtains the result, it should call one of the two callbacks:

Result	Call
Success	myResolve (result value)
Error	myReject (error object)

Promise Object Properties

A JavaScript Promise object can be:

- Pending
- Fulfilled
- Rejected

The Promise object supports two properties: **state** and **result**.
While a Promise object is "**pending**" (working), the result is **undefined**.
When a Promise object is "**fulfilled**", the result is a **value**.
When a Promise object is "**rejected**", the result is an **error object**.

Async / await

async makes a function return a Promise.
await makes a function wait for a Promise.

Async syntax

The keyword `async` before a function makes the function return a promise:

```
async function myFunction() {
            return "Hello";
}
```

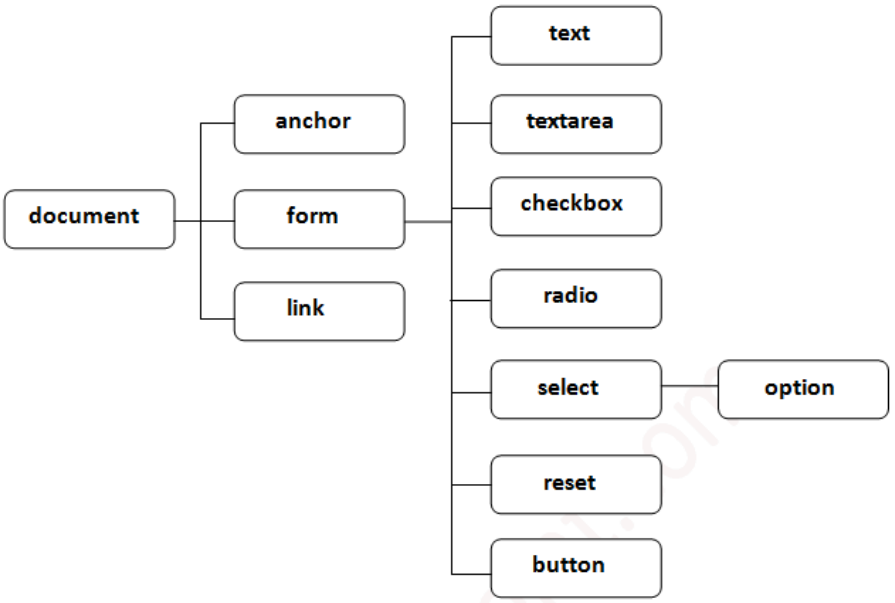
Await syntax

The `await` keyword can only be used inside an `async` function.
The `await` keyword makes the function pause the execution and wait for a resolved promise before it continues:

```
let value = await promise;
```

HTML DOM (Document Object Model)

With the HTML DOM, JavaScript can access and change all the elements of an HTML document.
When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.
The **HTML DOM** model is constructed as a tree of Objects.



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

What is the DOM?

The DOM is a W3C (World Wide Web Consortium) standard.
The DOM defines a standard for accessing documents:
"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

What is the HTML DOM?

The HTML DOM is a standard object model and programming interface for HTML. It defines:

- *The HTML elements as objects*
- *The properties of all HTML elements*
- *The methods to access all HTML elements*
- *The events for all HTML elements*

In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

DOM Programming Interface

The HTML DOM can be accessed with JavaScript (and with other programming languages).
In the DOM, all HTML elements are defined as **objects**.
The programming interface is the properties and methods of each object.
A **property** is a value that you can get or set (like changing the content of an HTML element).
A **method** is an action you can do (like adding or deleting an HTML element).

Example:-

```
<html>
<body>
      <p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>
</body>
</html>
```

In the example above, `getElementById` is a **method**, while `innerHTML` is a **property**.

The getElementById() Method

The most common way to access an HTML element is to use the `id` of the element.
In the example above the `getElementById` method used `id="demo"` to find the element.

The innerHTML Property

The easiest way to get the content of an element is by using the `innerHTML` property.
The `innerHTML` property is useful for getting or replacing the content of HTML elements.
The `innerHTML` property can be used to get or change any HTML element, including `<html>` and `<body>`.

Finding HTML Elements

Method	Description
document.getElementById(*id*)	Find an element by element id
document.getElementsByTagName(*name*)	Find elements by tag name
document.getElementsByClassName(*name*)	Find elements by class name
document.querySelectorAll(*name*)	Find elements by CSS Selector (id, class names, types, attributes, values of attributes, etc)

Changing HTML Elements

Property	Description
element.innerHTML = *new html content*	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element

Adding and Deleting Elements

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

Adding Events Handlers

Property	Description
document.getElementById(*id*).onclick = function(){*code*}	Adding event handler code to an onclick event

JSON

- JSON stands for **JavaScript Object Notation**
- JSON is a lightweight data-interchange format
- JSON is plain text written in JavaScript object notation
- JSON is used to send data between computers
- JSON is language independent

Why use JSON?

The JSON format is syntactically similar to the code for creating JavaScript objects. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.
Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.
JavaScript has a built in function for converting JSON strings into JavaScript objects:

```
JSON.parse()
```

JavaScript also has a built in function for converting an object into a JSON string:

```
JSON.stringify()
```

JSON Syntax Rules:

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Values

In JSON, *values* must be one of the following data types:

- a string
- a number
- an object
- an array
- a boolean
- null

In JavaScript values can be all of the above, plus any other valid JavaScript expression, including:

- a function

- a date
- undefined

Example:

```
{
      “firstName”: "John",
      “lastName”: "Doe",
      “age”: 50,
      “eyeColor”: "blue"
}
```

Regular Expression

A regular expression is a sequence of characters that forms a search pattern. The search pattern can be used for text search and text replace operations.

What is a regular expression?

A regular expression is a sequence of characters that forms a **search pattern**. When you search for data in a text, you can use this search pattern to describe what you are searching for.
A regular expression can be a single character, or a more complicated pattern. Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Syntax:

```
/pattern/modifiers;
```

Example:

```
/Hello/i;
```

/Hello/i is a regular expression
Hello is a pattern (to be use in a search)
i is a modifier. (modifies the search to be case-insensitive)

Regular Expression Modifier:

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

Regular Expression Pattern:

Expression	Description
[abc]	Find any of the character between the brackets
[0-9]	Find any of the digit between the brackets
(x|y)	Find any of the alternatives separated with |

Metacharacters:

Metacharacter	Description
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this : \bWORD, Or at the ending of a word like this: WORD\b
\uxxx	Find the unicode character specified by the hexadecimal number xxxx

Quantifier:

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero or more occurrences of n
n?	Matches any string that contains zero or one occurrences of n