GreatStack.

Watch course on YouTube: https://www.youtube.com/watch?v=Pyv0tMm5i_w

Table Content

Section - 5 (Functions)

- 5.1 Functions
  - -Parameters / Arguments
  - -return
- 5.2 Anonymous Functions
- 5.3 Recursive function
- 5.4 Default Parameters

Section - 6 (Objects & Prototype)

- 6.1 Object
  - -key
  - -Value
  - -method
- 6.2 Constructor functions
- 6.3 Prototype
- 6.4 Object Destructuring
- 6.5 Object literal syntax extensions

Section - 7 (Classes)

- 7.1 Class
- 7.2 Getters & Setters
- 7.3 Class Expression
- 7.4 Inheritance
- 7.5 Static Methods
- 7.6 Private Methods

Section - 8 (DOM - Document Object Model)

- 8.1 Node
  - -Text Node
  - -Element Node
  - -Child Node
  - -Parent Node
  - -Descendent Node
  - -Sibling Node
- 8.2 Query/Get Elements
- 8.3 Create / clone Element
- 8.4 Add node to document
- 8.5 Get Element Details
- 8.6 Modify Element
- 8.7 Get and Modify Element Class
- 8.8 Remove Node
- 8.9 event listener(.add/.remove)

Getting Started

What is JavaScript?

JavaScript is a scripting language that is used to create interactive and dynamic websites.

Interactive website means, user can perform some action on website for example: Button click, Submit a form, write comments and live chat.

Dynamic website means, the website that changes it's content or layout like: Sliding effect, Ecommerce website and Quiz website.

Why you should learn JavaScript?

JavaScript is the most popular programming language in the world. It is used in almost all popular websites like: Google, Facebook, Twitter, Amazon, Netflix and many others.

Great thing about JavaScript is that you will find tons of frameworks and Libraries to reduce your time to create websites and mobile apps. Some of the popular frameworks are: React, Angular and Vue.js

If you learn JavaScript, it opens up a lot of job opportunities in the software development industry.

What is the use of JavaScript?

Uses of JavaScript is not only limited to front-end web development, It is also used in back-end web development, Mobile app development, Desktop app development, Game Development and API creation.

Using JavaScript you can easily create website like Amazon and Netflix. Mobile apps like Instagram and WhatsApp. Games like Tic Tac Toe and Snake Games.

Basics

What is Variables in JavaScript?

Variables are used to store data. It is like a container, where we use to store things at home.

To declare a variable in JavaScript, we use the var, let, or const keyword.

```
var x;
```

Here 'var' is the keyword to declare a variable and 'x' is the identifier or name of the variable.

We can also declare variable like

```
let x;
```

Or

```
const x;
```

The var keyword is the oldest and most common way to declare variables

The let keyword is a newer keyword that was introduced in ES6. (We will learn more about ES6 later in advance JavaScript)

The const keyword is used to declare constants, which are variables that cannot be changed.

Once you have declared a variable, you can assign a value to it using the equal sign (=).

For example:

```
var x = 30;
```

the following code declares a variable called x and assigns it the value "30".

Variables can be used to store any type of data, including numbers, strings, objects, and arrays

Example:

```
var x = "Hello World"; //string
```

A JavaScript name must begin with:

A letter (A-Z or a-z)
A dollar sign ($)
Or an underscore (_)
Subsequent characters may be letters, digits, underscores, or dollar signs.
Numbers are not allowed as the first character in names.

JavaScript is case sensitive.

firstName and firstname are different.

Let keyword:

Let keyword in JavaScript is used to declare a block-scoped variable. This means that the variable is only visible within the block in which it is declared.

Example:

```
let x = 10;
if (x > 5) {
  let y = 20;
  console.log(y);  // 20
}
console.log(y);  // ReferenceError: y  is not defined
```

if you try to access the variable y outside of the if statement, you will get a ReferenceError. This is because the variable y is not defined outside of the if statement.

Const keyword:

The const keyword in JavaScript is used to declare a constant variable. This means that the variable cannot be reassigned to a new value.

Example:

```
const a =4;
console.log(a); // 4
a = 5 // Error: Cannot assign to constant variable
```

Data Types in JavaScript

Data types is an important concept in any programming language. To perform any task with the Data, it is import to know it's type. Data types in JavaScript are divided into 2 categories:

Primitive data type and **reference data type.**

There are 7 primitive data types in JavaScript, Which are:
string, number, boolean, null, undefined, bigint and symbol

- String :

In JavaScript, a string is a sequence of zero or more characters. A string starts and ends with either a single quote(') or a double quote (").

JavaScript strings are for storing and manipulating text.

Example:

```
let firstName = "Elon";  // Double quotes
let lastName = 'Musk';  // Single quotes
```

- Number :

Number represents integer and floating-point numbers.

Example:

```
let num = 100;
```

The following statement declares a variable and initializes its value with an integer

```
let marks = 96.5;
```

Here we have stored the floating-point number.

If we write

```
let marks = 96.00;
```

JavaScript will convert it into an integer number to use less memory.
It will be saved as **96**

```
console.log(marks); // 96
```

```
let x = 10; // type is number.
let x = "10"; // type is string.
```

To check the type of the data stored in a variable , we use typeof operator

Example:

```
console.log(typeof x) // string
```

-Boolean :

The boolean type has two values: **true** and **false**.
Example:

```
let learning = true;
let completed = false;
console.log(typeof completed) // boolean
```

```
let x = 20>10;
```

Result of this comparison is true, so the data type will be boolean.

```
console.log(x); // true
console.log(typeof x); // boolen
```

```
let x = 20<10;
console.log(x); // false
console.log(typeof x); // boolen
```

- Undefined :

If a variable is declared but the value is not assigned, then the value of that variable will be **undefined**.

And the data type is also **undefined**.

Now understand it with example:

```
let age;
console.log(age); // undefined
console.log(typeof age); // undefined
```

we have declared a variable **age** but not assigned any value.

- Null :

In the JavaScript, null is a special value that represents empty or unknown value

Example:

```
let number = null;

console.log(number) // null
console.log(typeof number) // object
```

The type should be "null" but it says the type is object.

It is a known bug in JavaScript.

JavaScript defines that **null** is equal to **undefined**

To check this, let me write:

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

** There are 2 other new primitive data-types, which are **Symbol** and **BigInt**, that we will study in our upcoming chapters.

Reference Data-Type:

So the first reference data type is **"Object"**:

In JavaScript, an object is a collection of properties, where each property is defined as a key-value pair.

```
let person = {};
console.log(typeof person); // object
```

The following example defines an empty object. Now I will create an object with some properties:

```
let person = {
  firstName: 'Elon',
  lastName: 'Musk',
  age : 35}
```

An object can store different data type. In this Person object, **firstName** and **lastName** are string and **age** is number.

- Array :

Arrays are a **type of object** that stores a collection of values.

For example,

```
let numbers = [1, 2, 3, 4, 5];
```

the following code creates an array of numbers:

```
console.log(numbers);
console.log(typeof numbers); // Object
```

- Function :

Functions are a **type of object** that can be used to execute code.

Example:

```
function msg() {
  console.log("Hello World");
}
console.log(typeof msg); // function
```

It says that the type of data is “function” but it is also an Object data-type.

-- **JavaScript is a dynamically typed language**.

So we can store different data-type in the same variable.

Example:

```
let x;
console.log(typeof x); // undefined

x = "GreatStack";
console.log(typeof x); // string

x = 100;
console.log(typeof x); // number

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

Operators In JavaScript

Operators in JavaScript are symbols, that are used to perform operations on operands.

Operands are the values and variables.

For Example:

```
console.log(10 + 20); // 30
```

Here (+) is an operator that performs addition, and 10 and 20 are operands.

Here is the List of Different Operators that we will learn in this course.

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

Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations on operands.

// Addition

```
let sum = 5 + 3;
console.log(sum); // 8
```

// Subtraction

```
let difference = 10 - 4;
console.log(difference); // 6
```

// Multiplication

```
let product = 2 * 6;
console.log(product); // 12
```

// Division

```
let quotient = 15 / 3;
console.log(quotient); // 5
```

// Remainder (Modulus)

```
let remainder = 17 % 4;
console.log(remainder); // 1
```

// Exponentiation

```
let result = 2 ** 4;
console.log(result); // 16
```

Assignment Operator:

Assignment operators are used to assign values to variables. We use (=) sign for assignment operator.

Example:

```
let x = 5;
```

We have multiple assignment operator like:

// Addition assignment

```
x += 3;
console.log("x:", x); // x: 8
```

// Subtraction assignment

```
x -= 2;
console.log("x:", x); // x: 6
```

// Multiplication assignment

```
x *= 4;
console.log("x:", x); // x: 24
```

// Division assignment

```
x /= 3;
console.log("x:", x); // x: 8
```

// Remainder (Modulus) assignment

```
x %= 5;
console.log("x:", x); // x: 3
```

// Exponentiation assignment

```
x **= 2;
console.log("x:", x); // x: 9
```

- Increment / Decrement:

The increment and decrement operators are used to increase or decrease the value of a variable by 1.

The increment operator is **(++)** and, the decrement operator is (**--**).

The increment and decrement operators can be used in two ways:
Prefix and Postfix.

Let's learn **Prefix** increment and decrement operators:

```
let a = 10;

console.log(++a); // 11
console.log(a); // 11
```

In this example operator is placed **before the variable**, and the value of the variable is incremented before it is used.

```
let a = 10;

console.log(--a); // 9
console.log(a); // 9
```

Let's learn **Postfix** increment and decrement operators:

```
let a = 10;

console.log(a++); // 10
console.log(a); // 11
```

In this example operator is placed **after the variable**, and the value of the variable is used before it is incremented.

```
let a = 10;
console.log(a--); // 10
console.log(a); // 9
```

Comparison operators:

Comparison operators compare two values and give back a boolean value: either **true** or **false**.

Comparison operators are useful in decision-making and loop programs in JavaScript.

Let's see some examples:

< (less than)
> (greater than)
<= (less than or equal to)
>= (greater than or equal to)
== (Equal checks)
!= (inequality) (not equal) (**flipped** value of equal checks)
===(strict equality checks) (checks the **Data type**)
!==(strict inequality (!==) (**flipped** value strict equality checks)

Example:

```
const a = 10;
const b = 20;

console.log(a < b); // true
console.log(a > b); // false
console.log(a <= b); // true
console.log(a >= b); // false

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

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

```
const a = "10";
const b = 10;
console.log(a == b); // true
console.log(a === b); // false
```

Logical Operator:

Logical operators perform logical operations like:

AND (&&), **OR** (||), **NOT (!)**.

Logical AND (&&):

Evaluates operands and return true only if **all are true**

true && true; // true
true && false; // false
false && true; // false
false && false; // false

```
let x = 5;
let y = 10;
console.log(x > 0 && y > 0); // true
console.log(x > 0 && y < 0); // false
console.log(x < 0 && y > 0); // false
console.log(x < 0 && y < 0); // false
```

Logical OR (||):

Returns true even if **one of the multiple operands is true**

true && true; // true
true && false; // true
false && true; // true
false && false; // false

```
let a = 5;
let b = 10;
console.log(a > 0 || b > 0); // true
console.log(a > 0 || b < 0); // true
console.log(a < 0 || b > 0); // true
console.log(a < 0 || b < 0); // false
```

Logical NOT (!):

Converts operator to boolean and **returns flipped value**

```
let Yes = true;
let No = false;
console.log(!Yes); // false
console.log(!No); // true
```

Operator Precedence:

Operator precedence in JavaScript determines the order in which, operators are parsed concerning each other.

example:

```
let result = 2 + 3 * 4;
console.log(result); // Output: 14
```

Why multiplication is performed before addition, it is because of the operator's precedence value. In below table you can see the precedence of multiplication is 12 and precedence of addition is 11.

Operator	Precedence
Exponentiation	13
Multiplication, Division, Modulus	12
Addition, Subtraction	11
Left Shift, Right Shift	10
Equality Operators	8
Bitwise AND	7
Bitwise XOR	6
Bitwise OR	5
Conditional (ternary) Operator	2
Assignment Operators	2
Comma	1

So the higher precedence is performed first. That's why it Is multiplying 3 and 4 after that it is adding 2 and the result is 14.

Operator Associativity:

Operator associativity in JavaScript defines the order in which operators of the same precedence are evaluated.

There are two types of operator associativity:

- Left-to-right
- And Right-to-left

Left-to-right associativity:

In Left-to-right associativity Operators are evaluated from left to right.

Example:

```
let result = 4 - 2 - 1;
console.log(result); // Output: 1
```

In this example only subtraction operator is used and it's associativity is from left to right as you can see in the table.

The expression 4 - 2 - 1 is evaluated from left to right.

First, 4 - 2 is calculated, resulting in 2, then, 2 - 1 is evaluated, giving us the final result of 1.

Right-to-left associativity:

In Right-to-left associativity Operators are evaluated from right to left.

For example,

```
let result = 2 ** 3 ** 2;
console.log(result); // Output: 512
```

In this example only exponentiation operator is used and it's associativity is from **Right to left** as you can see in the below table.

First, 3 ** 2 is calculated, resulting in 9. Then, 2 ** 9 is evaluated, giving us the final result of 512.

Operator	Associativity
Exponentiation	Right-To-Left
Multiplication, Division, Modulus	Left-To-Right
Addition, Subtraction	Left-To-Right
Left Shift, Right Shift	Left-To-Right
Bitwise AND	Left-To-Right
Bitwise OR	Left-To-Right
Bitwise XOR	Left-To-Right
Conditional (ternary) Operator	Left-To-Right
Equality Operators	Left-To-Right
Assignment Operators	Right-To-Left
Comma	Left-To-Right

Loops in JavaScript

In programming, loops are used to repeat a block of code.

For example, if you want to display a message 100 times, then you can use a loop.

First we will learn about
for loop:

Example 1: Display a Text 10 Times:

```
for (let i = 1; i <= 10; i++) {
  console.log("GreatStack");
}
```

Example 2: Display Numbers from 1 to 10

```
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

Example: 3:

```
let coding = ["JavaScript", "Python", "CPP"];
for (let i = 0; i < coding.length; i++) {
  console.log(coding[i]);
}
```

We can create another Loop in side a loop also.

example:

```
for (let i = 1; i <=5; i++) {
    console.log(i);

    for (let j = 1; j <= 3; j++) {
      console.log("Inner Loop: " + j);
    }
  }
```

Now we will learn
while Loop:

Syntax of while loop is:

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

1. A while loop evaluates the condition inside the parenthesis ().
2. If the condition evaluates to true, the code inside the while loop is executed.
3. The condition is evaluated again.
4. This process continues until the condition is false.
5. When the condition evaluates to false, the loop stops.

Example:

JavaScript code snippet uses a while loop to print numbers from 0 to 10.

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

JavaScript do...while Loop:

The syntax of **do...while loop** is:

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

Here,
1. The body of the loop is executed at first. Then the condition is evaluated.
2. If the condition evaluates to true, the body of the loop inside the do statement is executed again.
3. The condition is evaluated once again.
4. If the condition evaluates to true, the body of the loop inside the do statement is executed again.
5. This process continues until the condition evaluates to false. Then the loop stops.

Example: Display Numbers from 1 to 5

```
let i = 1;
do {
    console.log(i);
    i++;
} while(i <= 5);
```

Now let's understand when we will use for loop and when we will use while loop

A for loop is usually used when the number of iterations is known

And while and do...while loops are usually used when the number of iterations are unknown.

We have to repeat the code until the condition is false

So this was **for loop**, **while loop**, and **do.. while loop**.

Apart from this there are some other loops also like
- **For of**
- **For In**
That we will study in advance JavaScript.

Now we will learn about
break and continue statement:

The break statement is used to terminate the loop immediately.

```
for (let i = 1; i <= 5; i++) {
  // condition to break
  if (i == 3) {
      break;
  }
  console.log(i);
}
```

The continue statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.

```
for (let i = 1; i <= 5; i++) {
  // condition to continue
  if (i == 3) {
      continue;
  }
  console.log(i);
}
```

JavaScript Functions

In this tutorial, we will learn about **function** in JavaScript.

- A function is a block of code that performs the **specific task**.
- Functions in JavaScript are **reusable blocks of code** that can be called from anywhere in your program.

Here is the syntax to declare a function.

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

A function can be defined using the **function keyword**, followed by the **name of a function**. The body of function is written within curly braces {}.

=> Now let's **create a function** to print a Text

We will use the **function** keyword and the function name is Greet.

```
function greet() {
  console.log("Hello, GreatStack");
}
```

Now we need to call this function to execute the codes written in this function. For calling this function we will just write the function name and parenthesis.

```
greet() // Hello, GreatStack
```

=> Now we will learn about **Parameters** and **Arguments**:

Parameters are the **variables**, that are declared in the function definition,
while arguments are the **values,** that are passed to the function when it is called.

Let's see one example of **function with parameters:**

```
function greet(firstName, lastName) {
  console.log("Hello " + firstName + " " + lastName);
}
```

Here I am adding 2 parameters "**firstName**" and "**lastName**".
We can add multiple parameters in the function.

To call this function we will write:

```
greet("Elon", "Musk");
```

```
greet(100, 200); // we can pass any data type in the function
```

We can pass **less or more arguments** while calling a function.
If we pass **less arguments** then the rest of the parameters will become **undefined**.
If you pass more arguments then additional arguments **will be ignored**.

```
greet("Elon"); // Hello Elon undefined
greet("Mr", "Elon", "Musk"); // Hello Mr Elon
```

=> Now let's learn about **Default Parameters:**

Default parameters in JavaScript are parameters that **have a default value**. This means that, if the parameter is not passed to the function, the default value will be used.

Let's see one example of **Default Parameters**

```
function sum(x, y = 0) {
  console.log(x + y);
}

sum(10,15); // 25
sum(10); // 10
```

=> Now we will about **Function Return**

The **return statement** can be used to **return the value,** when the function is called.

The return statement denotes that the function has ended. Any code after **return** is not executed.

```
function add(a, b) {
  return a + b;
}
let result = add(10,15);
console.log("The sum is " + result); // The sum is 25
```

=> In JavaScript, a function can return **another function**.

Let's see one example:

```
function fn1(x) {

  function fn2(y)
  {
      return x * y;
  }
   return fn2;
}
var result = fn1(3);
console.log(result)
console.log(result(2));
```

=> Now we will learn **Callbacks in JavaScript:**

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

Let's understand the callbacks with one example:

```
function display(result) {
  console.log(result);
}

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

add(5, 10, display);
```

- When you pass a function as an argument, remember not to use parenthesis.

=> Now let's learn about **Anonymous functions**:

Anonymous functions in JavaScript, are functions that are not declared with a name.

Here is the syntax for the anonymous function:

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

Just write the function keyword and parenthesis then add your code in the curly braces

We can add the parameters also in anonymous function.

Let's understand with one example:

```
let sum = function (x, y) {
  return x + y;
};
```

Here we have declared one function and assigned this function in a variable.

This is know as function expression.

To call this function we will use "sum" variable;

And this function has 2 parameters, so we will pass 2 arguments.

```
let sum = function (x, y) {
  return x + y;
};
console.log(sum(5,10)); // 15
```

Let's see another example of Anonymous functions

```
(function () {
  console.log("Welcome to GreatStack");
})();
```

Here we have defines an anonymous function, and wraps it inside parentheses. Immediately after the function definition, **it is invoked by adding ().**

The function is executed immediately when the script runs, and it logs the message "**Welcome to GreatStack**" to the console.

Let's see another example of Anonymous functions:

```
setTimeout(function () {
  console.log("Welcome to GreatStack");
}, 5000);
```

setTimeout is an inbuilt method in JavaScript that accepts **one function** and **time in milliseconds**.

Here I am adding one a **Anonymous function** that display the message "**Welcome to GreatStack**" in console. And here I am adding 5000
So it will execute this function after 5000 milliseconds which is 5 seconds.

=> Now we will learn about **Recursive Functions:**

A recursive function in JavaScript is a function that **calls itself.**

```
function myFunction(){
  // Function code
    myFunction();
  }

myFunction();
```

Here myFunction is a recursive function, It is calling itself inside the function. A recursive function must have a condition to stop calling itself.

Otherwise, the function will be calling itself infinitely. To prevent infinite recursion, we can use If…else statements or any other conditional statement.

```
function myFunction(){
  if(condition){
    myFunction();
  }
  else{
      //  // stop calling recursion
  }
}
```

Now let's see one example: I want to print numbers in descending order.

```
function countDown(num){
  console.log(num);
  num--;
  if(num >= 0){
    countDown(num);
  }
}

countDown(10);
```

So this was all about recursive function, callback function and anonymous functions in JavaScript.

Objects in JavaScript

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

let's see the syntax to declare an Object.

const **object_name** = {
  key1: value1,
  key2: value2
}

We can use **let** keyword also, but it is good practice to declare objects with **const** keyword.

Let's see one example:

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35
};

console.log(typeof person); // object
```

Any object can store 2 things, which is

- **Properties** and
- **Methods**

In this **person** object, data is stored in **key:value** pairs.

firstName is Key and **Elon** is value
Age is key and **35** is value

These "**key: value**" pairs are called properties.

firstName: "Elon" ==> this is property of the **person** Object.

The **Key** is always stored as a **string**.

But in **value** we can add any type of data such as:
String, Number, Array, Boolean, Function and we can store an Object also .

When we **declare a function** as a value in **key:value** pair then it is known as **Methods**.

Let's learn how to **access the properties** of an Object:

We can access the value of a property by using its **key name**.

1. Using **dot Notation**

    objectName.key

Example:

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35
};

console.log(person.firstName);
```

2. Using **bracket Notation**

objectName["propertyName"]

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35
};

console.log(person["firstName"]);
```

When we add space in key name

Example: "**first Name**": "Elon",

Then we can't access the value using dot notation, In this condition we need to use bracket Notation only.

```
const person = {
  "first Name": "Elon",
  "last Name": "Musk",
  age: 35
};

console.log(person["first Name"]);
```

==> Let's see how to properties value:

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35
};

person.firstName = "Mr. Elon";

console.log(person.firstName);
console.log(person);
```

==> Add new properties in the Object:

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35
};

person.company = "Tesla";

console.log(person);
```

Deleting a property on an Object:

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35
};

delete person.age;

console.log(person);
```

==> Nested Objects

An object can also contain another object

For example:

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35,
  address: {
      street: "Tesla Road",
      city: "Austin",
      state: "Texas",
      country: "United States",
      zipCode: "78725"
  }
};

console.log(person.address.city);
```

Checking if a property exists:

To check if a property exists in an **object**, you use the **in** operator:

propertyName **in** objectName

It returns **true** if the propertyName exists in the objectName.

==> Now if you want to display all properties and values of an Object without knowing the property name then you can use **for…in Loop.**

for…in Loop:

The for...in loop allows you to access each property and value of an object without knowing the specific name of the property.

For example:

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35,
};

for(let prop in person){
  console.log(prop);
}
```

To **access the value** of each property we use, object[key]

```
const person = {
  firstName: "Elon",
  lastName: "Musk",
  age: 35,
};

for(let prop in person){
  console.log(prop +":"+ person[prop]);
}
```

There are multiple ways to create a JavaScript Object. This was the example of creating an object with object literal.

Let’s see another example where we will create an object with **new** keyword.

```
const person = new Object();
person.firstName = "Elon";
person.lastName = "Musk";
person.age = 35;
console.log(person);
```

- or -

```
const person = new Object({
  firstName : "Elon",
  lastName : "Musk",
  age : 35
});

console.log(person);
```

JavaScript Object Methods:

JavaScript **method** is an object **property** that contains a **function definition**.

Let's create one Object:

```
const person = {
    firstName : "Elon",
    lastName : "Musk",
    greet: function greet(){
        console.log("Hello World");
    }
}
person.greet();
```

```
const person = {
    firstName : "Elon",
    lastName : "Musk",
    greet: function(){ console.log("Hello World!"); }
}

person.greet();
```

Here we have used a **function expression** to **define a function** and **assigned** it to the **greet** property of the **person** object. Then calling the method using **greet()**

We can declare a function outside of the Object and assign it to an Object as a Method.

Here is one example:

```
const person = {
    firstName : "Elon",
    lastName : "Musk",
}

function greet(){
    console.log("Hello World!");
}

person.greeting = greet;

person.greeting();
```

Below code will log the function definition

```
console.log(person.greeting);
```

ES6 provides us one more way to define a method for an Object.

```
const person = {
    firstName : "Elon",
    lastName : "Musk",
    greet(){
        console.log("Hello World");
    }
}

person.greet();
```

JavaScript this Keyword:

To access the other properties of an object within a method of the same object, we can use "**this**" keyword.

Let's see the same example:

To access the **firstName** of the person object we use

```
person.firstName
```

But when we need to access the **firstName within the method**, we will use "**this**" keyword.

```
greet(){
        console.log("Hello " + this.firstName);
    }
```

So when we use "this" keyword within a **method**, it refers to the same **object**.

Now let's define a method that returns the full name of the person object by concatenating the first name and last name.

```
const person = {
    firstName : "Elon",
    lastName : "Musk",
    getFullName: function(){
        return this.firstName + " " + this.lastName;
    }
}
console.log(person.getFullName());
```

We can use **this** keyword outside of the method also, but it will refer to other object. If we use "**this**" keyword alone, or inside a function. Then it will refer to the **Global object**, that is: **window** object.

```
console.log(this)
```

When we use "this" key word in the "**Event**" then it will refer to the **element** that **received** the **event**.

"this" is a keyword, it is **not a variable**, so we **can't change the value** of "this".

JavaScript Constructor Function

In this tutorial, you'll learn about the JavaScript **constructor function** and how to use the **new keyword to create an object.**

In JavaScript, a **constructor function** is used to **create objects**.

Let's create an object:

```
const person = {
    firstName: 'Elon',
    lastName: 'Musk'
}
```

the following example create a person object with 2 properties, First Name and Last Name.
But this syntax is used when you have to create single object.

In programming, you often need to create many similar Objects like person Object.

To do that, you can use a constructor function.

Let me create on constructor function:

```
function Person () {
    this.firstName = 'Elon',
    this.lastName = 'Musk'
}
```

Constructor function is similar as a regular function but it is good practice to capitalize the first letter of your constructor function.

A constructor function should be called only with **new** operator.

We can use **new** operator to create an Object from a constructor function.

```
const person1 = new Person();
const person2 = new Person();
console.log(person1);
console.log(person2);
```

We can create multiple object of **same type** using a constructor function.

When a new Object is created using constructor function, "**this**" keyword refers to the newly created object. So the First Name and Last Name will **become the properties of** the newly created object.

In this example person1 and person2 object have same property values.

We can create multiple object of same type with different property value also using
Constructor Function Parameters:

```
function Person(fName, lName) {
    this.firstName = fName,
    this.lastName = lName
}

const person1 = new Person("Elon","Musk");
const person2 = new Person("Bill", "Gates");
console.log(person1);
console.log(person2);
```

Adding Properties And Methods in an Object:

Let's add one property in person1:

```
person1.age = 52;
console.log(person1);
console.log(person1.age);
console.log(person2.age); // undefined
```

Now let's add one method in person2:

```
person2.greet = function(){
    console.log("Hello, GreatStack");
}
person2.greet();
```

In this constructor function I have added only 2 properties, so let's see how to add a method in the constructor function.

```
function Person(fName, lName) {
    this.firstName = fName,
    this.lastName = lName,
    this.getFullName = function(){
        return this.firstName +" "+ this.lastName;
    }
}

const person1 = new Person("Elon","Musk");
const person2 = new Person("Bill", "Gates");
```

```
console.log(person1.getFullName());
console.log(person2.getFullName());
```

The problem with the constructor function is that,
when we create multiple instances of the Person,
then **getFullName() method** is duplicated in every instance, which is not memory efficient.

To resolve this, we can use the prototype so that all objects created from same constructor function can share the same methods.

So, Let's learn about:
JavaScript Object Prototype

In JavaScript, every **function** and **object** has its own **property** called prototype.

Let's create one object:

```
const person = {
    name: "Elon"
}
console.log(person);
```

This person object has one more property called prototype.

A prototype itself is also another object. So, the prototype has its own prototype. This creates a prototype chain.

Now let's learn:
Prototype Inheritance:

We can use the **Prototype** to add **properties and methods** to a constructor function. And objects **inherit** the properties and methods from a **prototype**.

So let's see the previous example of constructor function:

```
function Person(fName, lName) {
    this.firstName = fName,
    this.lastName = lName,
}

const person1 = new Person("Elon","Musk");
const person2 = new Person("Bill", "Gates");
```

Here we have only 2 properties in the constructor function.
So let's add one more property in the constructor function.

```
Person.prototype.gender = "Male";

console.log(person1);
console.log(person2);

console.log(person1.gender);
console.log(person2.gender);
```

Here we have added a new property **gender** to the **Person constructor function.**
Person1 and person2 objects inherits the gender property from constructor function.

This gender property is not available in the person1 and person2 object, but still we can access it.
Because it has been added in the prototype.
let's add one method in the constructor function:

```
Person.prototype.getFullName = function(){
    return this.firstName +" "+ this.lastName;
};

console.log(person1);
console.log(person2);

console.log(person1.getFullName());
console.log(person2.getFullName());
```

Here also you can see the **method** is not available in the person1 and person2 object. But it is available in it's prototype.

Now let's see what will happen if I change the prototype value.

```
function Person() {
    this.name = "Elon Musk"
}
Person.prototype.age = 25;

const person1 = new Person();
Person.prototype = {age : 52};
const person2 = new Person();
```

```
console.log(person1.age); // 25
console.log(person2.age); // 52
```

If a prototype value is changed, then all the new objects will have the changed property value.
All the previously created objects will have the previous value.

Object Destructuring:

Object destructuring in JavaScript is a feature that allows you to extract the properties of an object into variables. This can be useful for assigning the properties of an object to variables in a single statement.

```
const person = {
    firstName : "Elon",
    lastName: "Musk"
}
```

To store these properties value in a variable we use to write:

```
let fName = person.firstName;
let lName = person.lastName;
```

ES6 introduces the object destructuring syntax that provides an alternative way to assign properties of an object to variables:

Here is the syntax for object destructuring:

```
let { property1: variable1, property2: variable2 } = object;
```

Let's assign this object property on the variable using object destructuring:

```
let {firstName: fName, lastName: lName} = person;
```

If the **variables** have the **same names as the properties**

```
let {firstName, lastName} = person;
```

```
console.log(firstName);
console.log(lastName);
```

When you assign a property that does **not exist to a variable** using the object destructuring, the variable is set to **undefined**.

For example:

```
let {firstName, lastName, age} = person;
console.log(age); // undefined
```

Setting default values

```
const person = {
    firstName : "Elon",
    lastName: "Musk",
    age: 52
}
let {firstName, lastName, age = 18} = person;
console.log(age); // 52
```

Object Literal Syntax Extensions in ES6:

```
let fisrName = "Elon";
let lastName = "Musk";

const person = {
    fisrName,
   lastName
};

console.log(person);
```

So this is all about objects.

JavaScript Class

Classes are one of the features introduced in the ES6 version of JavaScript. JavaScript Class is a templates for creating Objects.

Here is the syntax for creating JavaScript class

```
class ClassName {
  constructor() {  }
}
```

To create a class, we use **class** keyword.

You should always add a **method** named **constructor()** in the class. The first letter of the class name should be in the capital letters.

JavaScript class is similar to the JavaScript constructor function.

To create a constructor function, we use to write:

```
function Person(name, age){
    this.name = name;
    this.age = age;
  }
const person1 = new Person("Elon Musk", 52);

console.log(person1);
```

now let's create the class.

```
class Person{
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
}
const person1 = new Person("Elon Musk", 52);
const person2 = new Person("Bill Gates", 67);

console.log(person1);
console.log(person2);
```

constructor() method initialize the properties of the object. JavaScript automatically calls the constructor() method when you create an object using Class.

Now let's learn about
JavaScript Class Methods

We can add any number of methods in JavaScript class.

```
greet(){
      return "Hello " + this.name;
     }

chnageName(newName){
      this.name = newName;
    }
```

```
console.log(person1);
console.log(person1.greet());
```

You can see the method has been added in the prototype of the Object.

```
person1.chnageName("Avinash");
console.log(person1);
```

Now let's Learn about
Getters and Setters in JavaScript:

Getters and setters are special methods in JavaScript that allow you to control how properties are accessed and modified.

They are defined using the **get** and **set** keywords.

Now let's understand the **getter method** with one example:

```
get greet(){
      return "Hello " + this.name;
      }

console.log(person1.greet);
```

Now let's understand the **setter method** with one example:

A setter is a method that is called when a property is modified. It can be used to do things like update the value of the property or perform some other action.

```
 set chnageName(newName){
      this.name = newName;
      }

person1.chnageName = "Avinash";
console.log(person1);
```

We can use the same method name as getter and setter.

Let's see one example:

```
get personName(){
      return this.name;
    }
set personName(newName){
      this.name = newName;
    }
const person1 = new Person("Elon Musk", 52);

console.log(person1.personName);

person1.personName = "Avinash";
console.log(person1.personName);
```

When we assign any value, It will **call the setter method** and **update** the property's value.

Now let's learn about the
JavaScript class expressions:

A class expression provides you an alternative way to define a new class. It is similar to a function expression, but it uses the class keyword instead of the function keyword.

If they are named, the name can be used to refer to the class later. If they are unnamed, they can only be referred by the variable that they are assigned to.

So let's define one class expression:

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

const person1 = new Person("Elon Musk");

console.log(person1);
```

A class expression doesn't require an identifier after the class keyword. And you can use a class expression in a variable declaration.

Now we will learn about
JavaScript Class Inheritance:

JavaScript class **Inheritance** allows you to **create new class** on the basis of already existing class.

Using class inheritance, a class can inherit all the methods and properties of another class. Inheritance is a useful feature that allows the code reusability.

To create a class inheritance, we use the **extends** keyword.

So let's create one class

```
class Person{
  constructor(name){
    this.name = name;
  }
  greet(){
    console.log("hello " + this.name);
  }
}
```

Now let's create another class that inherit all the methods of the Person class.

```
class Student extends Person{

}

const student1 = new Student("Peter")

student1.greet();
```

Now let's learn about
JavaScript super() method:

The **super()** method used inside a **child** class denotes its **parent** class.

```
class Student extends Person{
  constructor(name){
    super(name);
  }
}

const student1 = new Student("Peter")

student1.greet();
```

Here, the **super() method** inside **Student class** refers to the **Person class.**

When we will create a object using Student class,

this constructor method will be called automatically.

Then it will call **the super** and that **passes the arguments** to the person's constructor method.
That argument will be stored in the person's name.

Now let's understand
What is **Method or Property overriding or Shadowing method:**

If the parent class and child class has the same method or property name.
In this case, when we will call the method or property of an Object of the child class, It will override the method or property of the parent class. This is known as method overriding or shadowing method.

Let's understand the method overriding with one example:

```
class Person{
  constructor(name){
    this.name = name;
  }
  greet(){
    console.log("Hello Person " + this.name);
  }
}

class Student extends Person{
  greet(){
```

```
    console.log("Hello Student " + this.name)
  }
}

const student1 = new Student("Peter")

student1.greet();
```

Now we will learn about
JavaScript Static Methods:

Static methods are bound to a class, not the instances of that class. You can not call a static method on an object, It can be called only on the class.

```
class Person{
  constructor(name){
    this.name = name;
  }
  static greet(){
    console.log("Hello!");
  }
}

const person1 = new Person("Peter")

person1.greet() // Error (cannot call on object)

Person.greet();
```

If you want to use the object's properties inside the static method, then you can pass the object as a parameter:

```
class Person{
  constructor(name){
    this.name = name;
  }
  static greet(x){
    console.log("Hello " + x.name);
  }
}

const person1 = new Person("Peter")

Person.greet(person1);
```

So this was the static method in JavaScript.

Now we will learn about
JavaScript Private methods:

Private methods are accessible only within the class. It means we can not call the private methods outside of that class. By default, methods of a class are public.

To make the methods private, we need to start the method name with hash (# tag).

```
class Person{
  constructor(firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
  }
  #fullName(){
    return this.firstName + " " + this.lastName;
  }

  display(){
    console.log(this.#fullName());
  }
}

const person1 = new Person("Peter", "B")
person1.display();
```

We can not call fullName method outside of the class, we can only call the fullName method inside the class using **this** keyword.

Private static method example:

```
class Person{
  constructor(firstName, lastName){
    this.firstName = firstName;
    this.lastName = lastName;
  }
  static #fullName(x){
    return x.firstName + " " + x.lastName;
  }

  display(){
    console.log(Person.#fullName(this));
  }
}

const person1 = new Person("Peter", "B")
person1.display();
```

Access the private **static** method using class name **Person**.

Now we have completed classes in JavaScript.

DOM - Document Object Model

Now we will learn about
Document Object Model

The Document Object Model (DOM) is an *application programming interface* (API) for manipulating HTML documents.

The DOM provides functions that allow you to add, remove, and modify parts of the document effectively.

The DOM represents an HTML document as a tree of nodes.

```
<html>
    <head>
        <title>GreatStack</title>
    </head>
    <body>
        <p>JavaScript DOM</p>
    </body>
</html>
```

Now let's understand the Node Types:

```
<!DOCTYPE html>
<html>
    <head>
        <title>GreatStack</title>
    </head>
    <body>
        <!-- comment -->
        <p>JavaScript DOM</p>
    </body>
</html>
```

In this HTML code we have:
- Text Node
- Element Node
- Comment Node
- Document Type Node

Now let's learn about,
Node Relationships:

Any node has relationships to other nodes in the DOM tree, and it is same as described in the traditional family tree.

For example, <body> is a child node of the <html> node
<html> is the parent of the <body> node

<body> node is the **sibling** of the <head> node
because they share the same **immediate parent**, which is the **<html>** element.

```
        <div>
            <p>JavaScript DOM</p>
            <p>JavaScript DOM</p>
            <p>JavaScript DOM</p>
        </div>
```

 - Parent node
<p> - child node
All <p> are sibling

1- First Child
2- Last Child

2, 3 - next sibling
1, 2 - previous sibling

Selecting elements

Now let's learn about selecting elements using DOM:

To get elements in JavaScript there are various methods in DOM.

So the first one is:

JavaScript getElementById() method:

The **document.getElementById()** method returns an **Element object** that represents an **HTML element**

the getElementById() is only available on the document object.

```
 <div>
        <p id="message">Welcome to GreatStack</p>
</div>

<script>
   let msg = document.getElementById("message");
   console.log(msg);
</script>
```

If the document has **no element** with the specified **id**, then it will return null.

Because the id of an element is unique within an HTML document,
the document.getElementById() is a quick way to access an element.

If document has multiple elements with the same id, then it will return the first element.

Now let's learn about:
getElementsByName() method:

Every element on an HTML document may have a name attribute. Multiple HTML elements can share the same value of the **name** attribute like this:

```
        <p id="message">Welcome to GreatStack</p>
        <input type="radio" name="language" value="JavaScript">
        <input type="radio" name="language" value="Pyhon">
    </div>

    <script>
        let btn = document.getElementsByName("language");
        console.log(btn);
    </script>
```

Now we will learn about
getElementsByTagName() method:

The getElementsByTagName() method accepts a **tag name** and returns a live HTMLCollection of elements.

```
    <div>
        <p id="message">Welcome to GreatStack</p>
        <h1>first heading</h1>
        <h1>second heading</h1>
        <h1>third heading</h1>
        <h1>fourth heading</h1>
    </div>

 <script>
   let headings = document.getElementsByTagName("h1");
   console.log(headings);
 </script>
```

Now let's learn about
getElementsByClassName() method:

The getElementsByClassName() method returns an array-like of objects of the child elements with a specified class name.

The getElementsByClassName() method is available on the document element or any other elements.

```
<div>
    <p class="message">Welcome to GreatStack</p>
    <h1 class="message">first heading</h1>
    <div id="container">
        <h1 class="message">second heading</h1>
        <h1 class="message">third heading</h1>
    </div>
</div>

<script>
let msg = document.getElementsByClassName("message");
console.log(msg);

let cont = document.getElementById("container");

let cont_msg = cont.getElementsByClassName("message");
console.log(cont_msg);

</script>
```

Now let's learn about
JavaScript querySelector() and querySelectorAll() method :

The querySelector() method allows you to select the first element that matches one or more CSS selectors.

The querySelector() is a method of the Element interface.

```
let msg = document.querySelector(".message");
   console.log(msg);
```

you can use the querySelectorAll() method to select all elements that match a CSS selector or a group of CSS selectors:

The querySelectorAll() method returns a static NodeList of elements that match the CSS selector.

```
let msg = document.querySelectorAll(".message");
   console.log(msg);
```

We can use the query selector method on any element also

```
let cont = document.querySelector("#container");

let cont_msg = cont.querySelector(".message");
console.log(cont_msg);
```

Selectors:
Type selector: div, h1, h2, span, all tags
Class selector: .className
Id selector: #id
Attribute selector: [autoplay]
Grouping selectors: selector, selector, …
descendant combinator: ('div p')
Child combinator: selector > selector ("ul > li")
General sibling combinator: selector ~ selector
Adjacent sibling combinator: selector + selector
Pseudo-classes: "li:nth-child(2)"
Pseudo-elements: "p::first-line" or "p::before"

Now we will learn:
> **Traversing elements**:

To understand:-
how to Get the parent element.
how to Get child elements.
how to Get siblings of an element.

parentNode:
To get the parent node of a specified node in the DOM tree, we can use the parentNode property:

let parent = node.parentNode;

```
<div>
    <div class="title">
        <p class="text">Welcome to GreatStack</p>
    </div>
</div>
<script>
let txt = document.querySelector(".text");
console.log(txt.parentNode);
</script>
```

Getting Child Elements of a Node in JavaScript:

first child element, last child element, and all children of a specified element.

```
<div>
    <div class="title">
        <p>Welcome to GreatStack 1</p>
        <p>Welcome to GreatStack 2</p>
        <p>Welcome to GreatStack 3</p>
        <p>Welcome to GreatStack 4</p>
    </div>
</div>

<script>
let parent = document.querySelector(".title");

console.log(parent.firstChild);
console.log(parent.firstElementChild);

console.log(parent.lastChild);
console.log(parent.lastElementChild);

console.log(parent.childNodes);
</script>
```

Now we will learn
how to select the **next siblings**, **previous siblings** of an element:

```
<div>
    <div class="title">
        <p>Welcome to GreatStack 1</p>
        <p class="second">Welcome to GreatStack 2</p>
        <p>Welcome to GreatStack 3</p>
        <p>Welcome to GreatStack 4</p>
    </div>
</div>

<script>
let second = document.querySelector(".second");

console.log(second.previousElementSibling);
console.log(second.nextElementSibling);
</script>
```

Manipulating elements

Now let's learn how to **Manipulate the elements**:

createElement() :

To create an HTML element, we use the **createElement()** method:

The document.createElement() **accepts an HTML tag name** and returns a **new Node with the Element type**.

```
let div = document.createElement("div");
div.innerHTML = "<p>Welcome to GreatStack</p>";
console.log(div);
```

```
document.body.appendChild(div);
```

```
//adding an id to div
div.id = "title";
//adding a class name to div
div.className = "title";

console.log(div);
```

appendChild()

Use appendChild() method to add a node to the end of the list of child nodes of a specified parent node.

The appendChild() can be used to move an existing child node to the new position within the document.

```
<ul id="menu">
    <li>Home</li>
    <li>About</li>
    <li>Blog</li>
    <li>Project</li>
</ul>

<script>
let menu = document.getElementById("menu");

let list = document.createElement("li");
list.innerHTML = "Contact";

menu.appendChild(list);
</script>
```

textContent

To get the text content of a node and its descendants, you use the textContent property.

```
<ul id="menu">
    <li>Home</li>
    <li>About</li>
    <li>Blog</li>
    <li style="display:none">Project</li>
</ul>

<script>
let menu = document.getElementById("menu");
console.log(menu.textContent);
console.log(menu.innerText);

menu.textContent = "Hello!";

</script>
```

innerHTML:

```
<ul id="menu">
    <li>Home</li>
    <li>About</li>
    <li>Blog</li>
    <li>Project</li>
</ul>

<script>
let menu = document.getElementById("menu");
console.log(menu.innerHTML);

menu.innerHTML = "<h1>hello</h1>"
// menu.textContent = "<h1>hello</h1>";
</script>
```

> after() method

We can use after() method to insert **one or more nodes** after the **element**.

```
Element.after(node)
```

```
Element.after(node1, node2, ... nodeN)
```

after() method also accepts one or more strings.

```
Element.after(str1, str2, ... strN)
```

> append() method

We can use append() method to insert a set of Node objects or String objects after the last child of a parent node.

```
parentNode.append(newNodes);
```

append() ----> accept multiple nodes
Append() ----> accept strings also

appendChild() -----> a single node
appendChild() -----> no string (only node).

> prepend() method

The prepend() method inserts a set of Node objects or DOMString objects before the first child of a parent node:

```
parentNode.prepend(NewNodes);
```

> insertAdjacentHTML() method

The prepend() method inserts a set of Node objects or DOMString objects before the first child of a parent node:

```
element.insertAdjacentHTML(positionName, text);
```

'beforebegin': before the element
'afterbegin': before its first child of the element.
'beforeend': after the last child of the element
'afterend': after the element

Text:
'
Contact Us
'

HTML example:

```
<ul id="menu">
    <li>Home</li>
    <li>About</li>
    <li>Blog</li>
    <li>Project</li>
</ul>
```

> replaceChild() method

We can use replaceChild() method to replace child element by a new one.

```
parentNode.replaceChild(newChild, oldChild);
```

> cloneNode() method

We can use cloneNode() method to clone an element.

```
let newNode = originalNode.cloneNode();
```

only the original node will be cloned. All the node's descendants will not be cloned.

```
let newNode = originalNode.cloneNode(true);
```

the original node and all of its descendants are cloned.
> removeChild() method

To remove a child element of a node, we can use the removeChild() method

```
parentNode.removeChild(childNode);
```

```
menu.removeChild(menu.lastElementChild);
```

> insertBefore() method

We can use insertBefore() method to insert a new node before an existing node as a child node of a parent node.

```
parentNode.insertBefore(newNode, existingNode);
```

```
menu.insertBefore(newNode, menu.firstElementChild);
```

Now we will learn about
Attribute methods

Example:

```
<input type="text" id="username" placeholder="Enter your username">
```

The input element has two attributes:

1. The **type** attribute with the value **text**.
2. The **id** attribute with the value **username**.

//code
Element.attributes

Provide the **list of attribute** on that element.

```
let inputBox = document.getElementById("username");
```

Attribute methods:

element.**getAttribute**(name) – get the attribute value
element.**setAttribute**(name, value) – set the value for the attribute
element.**hasAttribute**(name) – check for the existence of an attribute
element.**removeAttribute**(name) – remove the attribute

getAttribute() method

```
let inputBox = document.getElementById("username");
console.log(inputBox.getAttribute("id"));
```

setAttribute() method

```
inputBox.setAttribute("class", "user");
console.log(inputBox);
```

hasAttribute() method

```
console.log(inputBox.hasAttribute("placeholder"));
```

removeAttribute() method

```
inputBox.removeAttribute("class");
console.log(inputBox);
```

Now we will learn about
Manipulating Element's Styles:

style property - get or set inline styles of an element.

Example
:

```
console.log(inputBox.style); // return all inline style styles

console.log(inputBox.style.backgroundColor);
// return background color value from inline style.
```

Example: setting background color in inline style
Code:

```
inputBox.style.backgroundColor = blue;
```

Using **cssText**:

```
inputBox.style.cssText = 'width:200px;background-color:red';
```

--> it completely override the inline css

```
inputBox.style.cssText += 'width:200px;background-color:red';
```

--> it will concatenate the new CSS property.

getComputedStyle() method

window object

// syntax

```
window.getComputedStyle(element, pseudoElement);
```

Example:

```
console.log(window.getComputedStyle(inputBox).width);
```

Now we will learn about
className property:

Tt reruns list of classes separated by space

```
<h1 id="title" class="main message">GreatStack</h1>
```

```
<script>
        let title = document.getElementById("title")
        console.log(title.className);
    </script>
```

Add new class using ClassName:

```
title.className = "new"; // it will override
```

```
 title.className += " new"; // it will concatenate
```

Now we will learn about
classList property: return collection of classes []

1- Getting the classes of an element:

```
console.log(title.classList);
```

2 - Add one or more classes:

```
title.classList.add("new", "new2");
console.log(title.className);
```

3 - Remove one or more classes:

```
title.classList.remove("main", "message");
console.log(title.className);
```

4 - replace a class:

```
title.classList.replace("message", "new-message");
console.log(title.className);
```

5 - check if an element has a specified class:

```
console.log(title.classList.contains("message"));
```

6 - Toggle a class:

```
title.classList.toggle("message"); // removed
title.classList.toggle("msg"); // added
console.log(title.className);
```

JavaScript Events

An event is an **action** that occurs in the web browser.

When we **click** on the web page, that is a **click event**
When we **move mouse cursor** on the web page, that is **mouse move event**.
When the **web page loads**, that is a **load event**.

Like that we have multiple types of Events in JavaScript.

Let's create some HTML elements.

```
<div class="container">
 <button type="button">Click here</button>
</div>
```

<button> is the target element

Event Flow:

There are 2 event models

Event bubbling - event starts at the most specific element and flows towards least specific element.
button --- to ---> document or up-to window object

Event capturing - event starts at the least specific element and flows toward the most specific element.
document --- to -----> button

We may **respond** to these events by **creating event handlers**.

An **event handler** is a **piece of code** that will be **executed when the event occurs**.

An **event handler** is also known as an **event listener**.

There are 3 ways to assign event handlers:

1) **HTML Event Handler Attributes**:

Event handler typically have names that begin with **on**.

Example:-

Event : click
Event handler : onclick

So let's assign this **onclick** event handler as HTML attribute.

```
<button id="btn" onclick="console.log('Button clicked!')">Click
here</button>
```

HTML event handler attribute can call a function also.
Example:

```
<button id="btn" onclick="displayMsg()">Click here</button>
    <script>
            function displayMsg(){
                console.log('Button Clicked');
            }
    </script>
```

When the **event occurs**, the web browser **passed an Event object** to the **event handler**:

```
<button id="btn" onclick="console.log(event)">Click here</button>
```

We can access the event objects properties like “type”

```
<button id="btn" onclick="console.log(event.type)">Click here</button>
```

```
<button id="btn" onclick="console.log(event.target)">Click here</button>
```

this keyword inside the event handler refers to **target element**.

```
<button id="btn" onclick="console.log(this)">Click here</button>
```

```
<button id="btn" onclick="console.log(this.id)">Click here</button>
```

event handler can access the element’s properties, for example:

```
<button id="btn" onclick="console.log(id)">Click here</button>
```

2) <u>DOM Level 0 event handlers</u>:

```
<button id="btn">Click here</button>
```

```
    let btn = document.getElementById("btn");
    btn.onclick = function(){
        console.log("Button Clicked!");
    }
```

this keyword refers to the target element.

```
   let btn = document.getElementById("btn");
    btn.onclick = function(){
        console.log(this.id);
    }
```

To remove the event handler

set the value of the **event handler property** to **null**:

```
btn.onclick = null;
```

3) DOM Level 2 event handlers:

DOM Level 2 Event Handlers provide two main methods

- **addEventListener()** – register an event handler
- **removeEventListener()** – remove an event handler

>>>> The **addEventListener()** method

the addEventListener() method accepts **three arguments**: an **event name**, an e**vent handler function**, and a Boolean value

addEventListener(event, function, useCapture);

For useCapture, the default value is false.
We can simply write:

addEventListener(event, function);

```
    btn.addEventListener('click', function(){
        console.log("button clicked!");
    })
```

Above example with Anonymous function.

Example with passing event Object:

```
btn.addEventListener('click', function(event){
        console.log(event.type);
    })
```

We can also refer to an external "named" function.

```
    let displayMsg = function(){
        console.log("button clicked!");
    }
    btn.addEventListener('click', displayMsg);
```

>>> The **removeEventListener()** method:

The removeEventListener() removes an event listener that was added using the addEventListener().

you need to pass the **same arguments** which was passed to the addEventListener()

```
btn.addEventListener('click', displayMsg);
btn.removeEventListener('click', displayMsg);
```

Now let's know about some useful JavaScript events:

mousemove - event fires repeatedly when you move the mouse cursor around the element

mousedown - when you **press** the mouse button on the element

mouseup - when you **release** the mouse button on the element.

mouseover - when the cursor move from outside to inside the boundaries of the element

mouseout - when the cursor is over an element and then moves another element.

keydown – fires when you press a key on the keyboard and fires repeatedly while you're holding down the key.

keyup – fires when you release a key on the keyboard.

keypress – fires when you press a character keyboard like a,b, c …. fires repeatedly while you hold down the key on the keyboard.

Scroll -- When you scroll a document or an element, the scroll events fire.

>> we will see more examples of events, when will create JavaScript projects.

Now we have c**ompleted the fundamentals of JavaScript**, In the next course we will learn **ES6** and **advance JavaScript** with 30+ projects.