


Learn JavaScript

World's Most Loved and Most Hated Language

Welcome To The Confusing World Of Javascript

A large, bold, black 'JS' logo is centered on a solid yellow rectangular background. The letters are thick and stylized, with the 'J' having a curved bottom and the 'S' being a simple, bold curve.

Introduction to JavaScript

JavaScript is a high level,object oriented,multi paradigm programming language.



JavaScript is a high-level, dynamic, object-oriented programming language that is primarily used to create interactive effects within web browsers. It was initially created to add interactivity to static HTML pages, but now it is used in a wide range of applications, including building web and mobile applications, creating desktop applications, developing server-side applications, and even creating robots and IoT devices.

JavaScript is a client-side scripting language, which means that it runs on the user's web browser, rather than on the server. It allows developers to create complex applications that can manipulate web page content, respond to user actions, and communicate with servers.

Some of the features of JavaScript include:

Dynamic typing: variables can hold values of any data type.

Functions as first-class

objects: functions can be treated like any other value.

Event-driven programming: code can respond to user actions or other events.

Asynchronous programming: code can execute tasks in the background while other code runs.

Object-oriented programming: code can be organized into objects that contain data and functions.

JavaScript has a large and active developer community, which has created a vast ecosystem of libraries, frameworks, and tools. Some popular JavaScript frameworks and libraries include React, Angular, and Vue.js, while popular tools include Node.js, Webpack, and Babel.

Why learn JavaScript

There are many reasons to learn JavaScript. Here are a few of the most important ones: JavaScript is a versatile language:

1. JavaScript is used not only for front-end web development, but also for back-end development, desktop application development, mobile app development, game development, and more. Learning JavaScript will allow you to build a wide range of applications.

2. JavaScript is in high demand: JavaScript is one of the most widely used programming languages in the world, and there is a high demand for developers who are skilled in it. This means that learning JavaScript can open up many job opportunities.

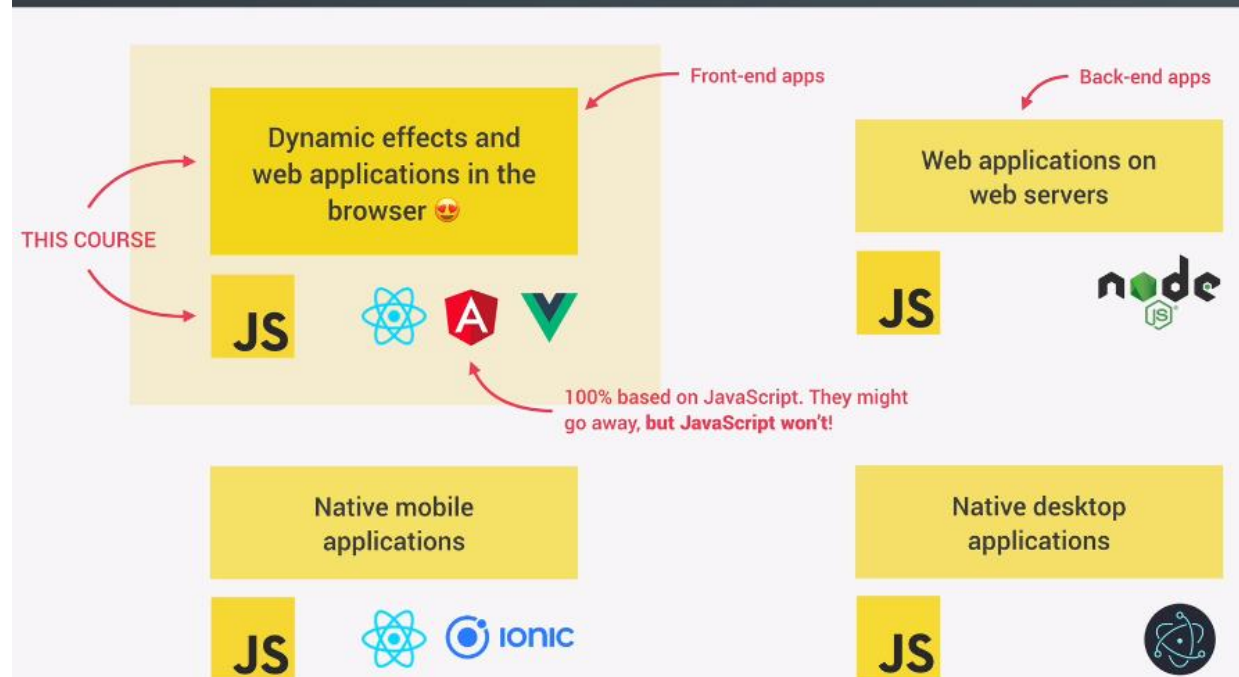
3. JavaScript is easy to learn: JavaScript has a simple syntax and is easy to get started with. You don't need a lot of experience to start building simple applications in JavaScript.

4. JavaScript is constantly evolving: JavaScript is a dynamic language that is constantly evolving. New libraries, frameworks, and tools are being developed all the time, which means there is always something new to learn and explore.

5. JavaScript is widely supported: JavaScript is supported by all major web browsers, which means that applications written in JavaScript can be used by a wide range of users.

Overall, learning JavaScript is a great investment in your future as a developer, as it is a powerful and versatile language that is in high demand.

THERE IS NOTHING YOU CAN'T DO WITH JAVASCRIPT (WELL, ALMOST...)



JavaScript , ECMAScript International & ECMAScript

ECMAScript is a JavaScript standard intended to ensure the interoperability of web pages across different browsers. It is standardized by Ecma International in the document ECMA-262. ECMAScript. Paradigm. Multi-paradigm: prototype-based, functional, imperative.

ECMAScript Editions

Ver	Official Name	Description
ES1	ECMAScript 1 (1997)	First edition
ES2	ECMAScript 2 (1998)	Editorial changes
ES3	ECMAScript 3 (1999)	Added regular expressions Added try/catch Added switch Added do-while
ES4	ECMAScript 4	Never released
ES5	ECMAScript 5 (2009) Read More	Added "strict mode" Added JSON support Added String.trim() Added Array.isArray() Added Array iteration methods Allows trailing commas for object literals
ES6	ECMAScript 2015 Read More	Added let and const Added default parameter values Added Array.find() Added Array.findIndex()
	ECMAScript 2016 Read More	Added exponential operator (**) Added Array.includes()
	ECMAScript 2017 Read More	Added string padding Added Object.entries() Added Object.values() Added async functions Added shared memory Allows trailing commas for function parameters
	ECMAScript 2018 Read More	Added rest / spread properties Added asynchronous iteration Added Promise.finally() Additions to RegExp
	ECMAScript 2019 Read More	String.trimStart() String.trimEnd() Array.flat() Object.fromEntries Optional catch binding
	ECMAScript 2020	The Nullish Coalescing Operator (??)

Types of JavaScript

1.Internal JavaScript

```
<script>  
  
</script>
```

2.External JavaScript

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

Console.log()

console.log is a method in JavaScript that outputs messages to the console. It's commonly used for debugging and testing purposes to display information about the execution of a program. The console.log method takes one or more arguments, which can be any JavaScript value or object. It then prints the value of those arguments to the console.

```
console.log("Hello Javascript");
```

Running JavaScript

1.Console

2.Browser

3.Node

Single Quotes Vs double Quotes

In JavaScript, both single quotes ('...') and double quotes ("...") can be used to represent strings. They are interchangeable and there is no functional difference between the two.

```
console.log("Hello Javascript");  
console.log('Hello Javascript');
```

Comment

JavaScript comments can be used to explain JavaScript code, and to make it more readable. JavaScript comments can also be used to prevent execution, when testing alternative code.

```
// single line comment  
  
/* this is a  
multi  
line comment */
```

Rules To Define Variable

In JavaScript, there are a few rules to follow when declaring and using variables:

1. Variable names must begin with a letter, underscore (_), or dollar sign (\$). They cannot begin with a number.
2. Variable names are case sensitive. For example, "myVariable" and "myvariable" are two different variables.
3. Variable names should be descriptive and meaningful. This makes your code easier to read and understand.
4. You cannot use reserved keywords as variable names. For example, "var", "function", and "if" are reserved keywords and cannot be used as variable names.
5. Variables can be declared using the "var", "let", or "const" keywords. "var" is the older way to declare variables, while "let" and "const" were introduced in newer versions of JavaScript. "let" is used to declare variables that can be reassigned, while "const" is used to declare variables that cannot be reassigned.
6. When declaring variables, it is good practice to use camelCase notation. For example, "firstName" instead of "first_name".
7. It is also good practice to declare all variables at the beginning of a function or block of code, to avoid issues with variable hoisting and to make your code more readable.

Variable Assignment and Declaration

In JavaScript, variables can be declared and assigned values using the var, let, or const keywords. Variable declaration refers to the act of creating a new variable with a given name. Variable assignment, on the other hand, is the act of assigning a value to a variable that has already been declared.

```
var variableOne;
```

```
let variableTwo;
```

```
const PI;
```

keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES

Scope

Global Scope: Variables declared outside of any function or block have global scope, which means they can be accessed and modified from anywhere in the code, including inside functions and blocks.

Local Scope: Variables declared inside a function or block have local scope, which means they can only be accessed and modified within that function or block.

Block scope, on the other hand, refers to the visibility and accessibility of variables declared inside a block, which is a set of statements enclosed in curly braces ({}), such as an if statement or a loop.

Function scope refers to the visibility and accessibility of variables declared inside a function in JavaScript. Variables declared inside a function have local scope, which means they can only be accessed and modified within that function.

Prefer camelCase

CamelCase is a naming convention used in programming and refers to the practice of writing compound words or phrases in which the first word is in lowercase and subsequent words are capitalized. This style of naming is often used for naming variables, functions, and objects in many programming languages, including JavaScript.

```
let firstPerson="shivang";
let collegeBranch="IT";
```


Data Types

JavaScript has 8 Datatypes

Primitive Data Type (Stack)

1. String
2. Number
3. BigInt
4. Boolean
5. Undefined
6. Null
7. Symbol

Non-Primitive Data Type (heap)

The Object Datatype

The object data type can contain:

1. An object
2. An array
3. A date

```
let dataExample = 100;
let dataExample = "string";
let dataExample = true;
let dataExample = undefined;
let dataExample = null;
let dataexample = 12113211136132132121211n;
const dataExample = {
  name: "shivang",
  job: "front-end developer",
};
const dataExample = [10, 20, 30, 40];
```

Temple literal

Template literals are literals delimited with backtick (``) characters, allowing for multi-line strings, string interpolation with embedded expressions, and special constructs called tagged templates.

```
let variableOne=100;  
let variableTwo="variable string";  
  
console.log(`we can use variable inside template  
literal ${variableOne} ${variableTwo}`);
```

Operators in JS

There are different types of JavaScript operators:

Arithmetic Operators

Assignment Operators

Comparison Operators

Logical Operators

Conditional Operators

Type Operators

Arithmetic Operators

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

```
//arithmetic operator
let numberOne=10;
let numberTwo=30;

console.log(numberOne + numberTwo);
console.log(numberOne - numberTwo);
console.log(numberOne * numberTwo);
console.log(numberOne / numberTwo);
console.log(numberOne % numberTwo);
console.log(numberOne ** numberTwo);
```

Assignment Operators

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

```
// assignment operator
let person="shivang";
console.log(person);
```

Comparison Operators

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

```
//comparison operator
let comOne=1;
let comTwo=10;

console.log(comOne==comTwo);
console.log(comOne===comTwo);
console.log(comOne!=comTwo);
console.log(comOne!==comTwo);
console.log(comOne>comTwo);
console.log(comOne<comTwo);
console.log(comOne>=comTwo);
console.log(comOne<=comTwo);
```

Logical Operators

Operator	Description
&&	logical and
	logical or
!	logical not

```
//logical operator
let logOne=10;
let logTwo=20;

// logical AND
if(logOne===10 && logTwo===20){
    console.log("yes this is true");
}

// logical OR
if(logOne===10 || logTwo===20){
    console.log("yes this is true");
}

//logical NOT
if(!(logOne===10)){
    console.log("yes this is true ");
} else {
    console.log("no this is not true");
}
```

Conditional Operators

```
//ternary operator
let num = 10;
let result = num > 5 ? "Greater than 5" : "Less than or equal to 5";
console.log(result); // Output: "Greater than 5"
```

Type Operators

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

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

Single Vs Double Vs Triple Equal To

= VS == VS ===

Double equal to operator (==): This operator compares the value on the left with the value on the right, and returns true if they are equal, even if they have different types. If the types are different, JavaScript tries to convert them to a common type before comparison.

Triple equal to operator (===): This operator compares the value on the left with the value on the right, and returns true if they are equal and have the same type.

```
a=1  
//single equal to assign value  
  
if(a==1){  
    //double equal to just compare value  
}  
  
if(a===1){  
    //triple equal to strict compare value  
}
```

Type Conversion

JavaScript variables can be converted to a new variable and another data type:

1.By the use of a JavaScript function

2.Automatically by JavaScript itself (coercion)

```
"1" + 1;  
"1" - 1;  
"1" / 2;  
"1" * 2;  
"string" * 1;
```

```
console.log(Number(true));  
console.log(Number(false));  
console.log(Number("10"));  
  
console.log(String(10));  
console.log(String(true));  
  
// true and false values  
// false values are 0 " " undefined null NaN  
console.log(Boolean(1));  
console.log(Boolean(0));  
console.log(Boolean(null));  
console.log(Boolean(NaN));
```

Post & Pre Operator

```
> let a=1;  
   let b = a++;  
   console.log(b);
```

1

```
<< undefined
```

```
> let a=1;  
   let b = ++a;  
   console.log(b);
```

2

```
> let a = 1;  
   let b = a--;  
   console.log(b)
```

1

```
<< undefined
```

```
> let a = 1;  
   let b = --a;  
   console.log(b)
```

0

Alert and prompt

In JavaScript, the `alert()` function is used to display a message to the user in a dialog box. In JavaScript, the `prompt()` function is used to display a dialog box that allows the user to enter some text.

```
alert("this is alert danger");  
  
prompt("this is used to take input");
```

Strict Mode

"use strict"; Defines that JavaScript code should be executed in "strict mode". The "use strict" directive was new in ECMAScript version 5. It is not a statement, but a literal expression, ignored by earlier versions of JavaScript. The purpose of "use strict" is to indicate that the code should be executed in "strict mode". With strict mode, you can not, for example, use undeclared variables.

```
"use strict";
```

If else

In JavaScript, if...else statements are used to control the flow of a program based on a condition.

```
let password="shivang123";  
let reEnter="shivang123";  
  
if(password===reEnter){  
  console.log("Password match");  
} else {  
  console.log("Password doesn't match");  
}
```


Switch

The switch statement in JavaScript is a control flow statement that allows you to test a variable or expression against multiple cases and execute different code blocks depending on the value of the variable. It provides a simpler way to write multiple if-else statements that test the same variable.

```
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("Today is Monday");
    break;
  case "Tuesday":
    console.log("Today is Tuesday");
    break;
  case "Wednesday":
    console.log("Today is Wednesday");
    break;
  default:
    console.log("Today is not Monday, Tuesday, or Wednesday");
}
```

Local and Global Variable

A variable declared outside of any function or block is known as a global variable. Global variables can be accessed from anywhere within a JavaScript program.

A variable declared inside a function or a block is known as a local variable. Local variables are only accessible within the function or block they were declared in.

```
//a,b,c are global variable
let a = 10;
let b = 20;
const c = 30;
```

Loops in JS

In JavaScript, loops are used to execute a block of code repeatedly until a specific condition is met. There are three main types of loops in JavaScript:

1.for loop

2.while loop

3.do-while loop

for loop: The for loop is used to execute a block of code a specific number of times. It consists of three parts: initialization, condition, and increment/decrement.

while loop: The while loop is used to execute a block of code as long as a specified condition is true.

do...while loop: The do...while loop is similar to the while loop, but it always executes the block of code at least once, even if the condition is false.

```
//while loop

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

//for loop

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

//do while loop

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

Types Of Error in Javascript

In JavaScript, there are several types of errors that can occur:

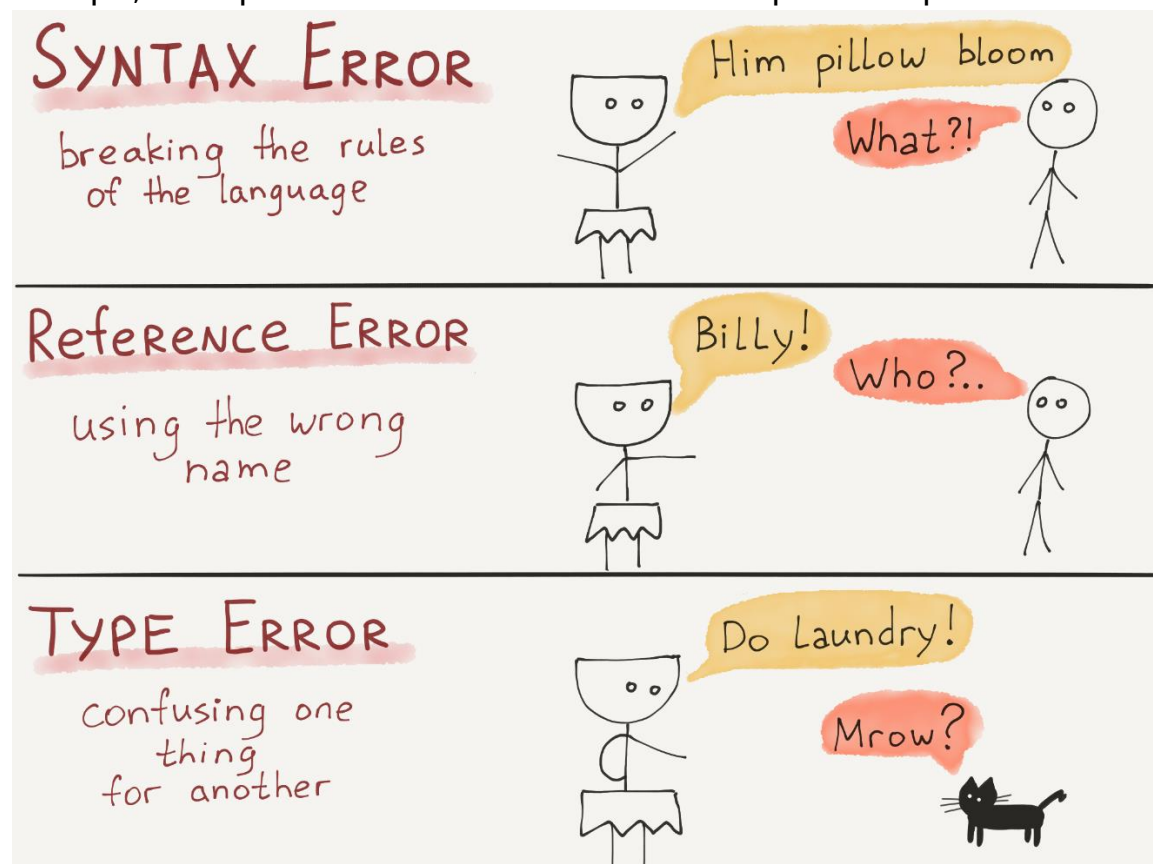
Syntax errors: These are errors that occur when the code is not written correctly according to the rules of the JavaScript language. For example, forgetting to close a bracket or quotation mark.

Runtime errors: These are errors that occur when the code is being executed, and something unexpected happens. For example, trying to access a property of an undefined object.

Logical errors: These are errors that occur when the code does not behave as intended. For example, if the logic of a loop is incorrect, and it does not iterate the correct number of times.

Type errors: These occur when a value is not of the expected type. For example, trying to perform arithmetic operations on non-numeric values.

Network errors: These are errors that occur when a network request fails. For example, if a request to an API does not return the expected response.



Function in JS

In JavaScript, a function is a block of code that performs a specific task and can be called by other parts of the code. Functions are an essential part of JavaScript programming, and they allow developers to reuse code and make their code more organized and easier to maintain.

1.Function Declaration

2.Function Expression

Named or Normal Function

Anonymous Functions

Arrow Function

Declaration & Expression

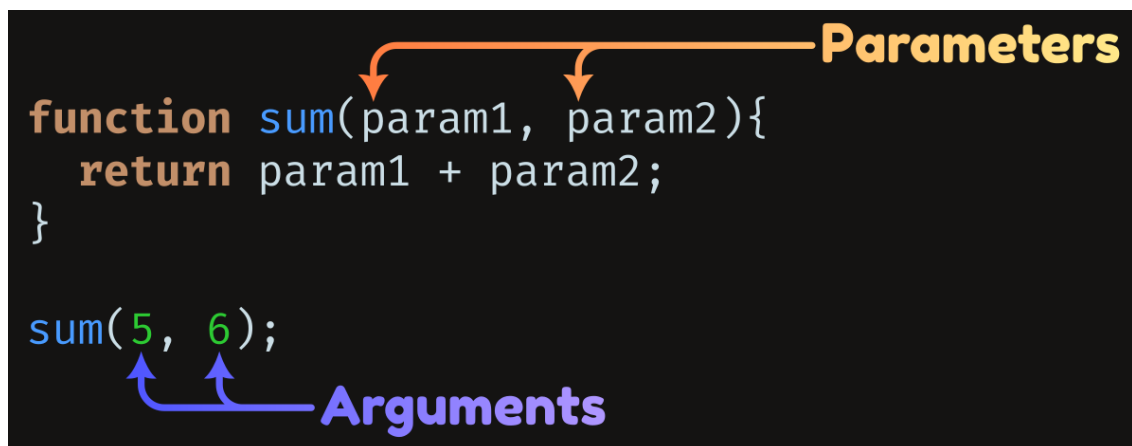
A function declaration is a way to define a function with the function keyword. This type of function is hoisted to the top of the current scope, which means that it can be called before it is defined in the code.

A function expression is a way to define a function as a value of a variable. This type of function is not hoisted to the top of the current scope, which means that it must be defined before it is called in the code.

Parameter and Argument in Function

In JavaScript, a parameter is a variable that is declared as part of a function's definition, and it represents a value that is expected to be passed into the function when it is called.

An argument, on the other hand, is the actual value that is passed into a function when it is called. In other words, an argument is a specific value that is provided to a function when it is invoked.



Call Function / Running Function / Execute Function / Invoke Function/
All Are Same Words having Same Meaning and Can Be used Interchangeable

```
function myfirstFunction() {  
  console.log("this is my first function");  
}  
myfirstFunction();  
  
function addTwo(a, b) {  
  let sum = a + b;  
  console.log(sum);  
}  
  
addTwo(2, 4);  
  
function addThree(a, b, c) {  
  let sum = a + b + c;  
  return sum;  
}  
  
const result = addThree(2, 2, 2);  
console.log(result);  
  
//Anonymous Functions  
const name = function (first, second) {  
  const firstName = first;  
  const lastName = second;  
  const fullName = `Your fullname is ${firstName + lastName}`;  
  return fullName;  
};  
  
console.log(name("shivang", "ramola"));
```

```
//arrow function
const twotimes = (EnterNumber) => EnterNumber * 2;

console.log(twotimes(8));

const threeTimes = (oneNumber, twoNumber) => {
  let result = oneNumber + twoNumber;
  console.log("this is executed before return");
  return result;
};
console.log(threeTimes(2, 5));
```

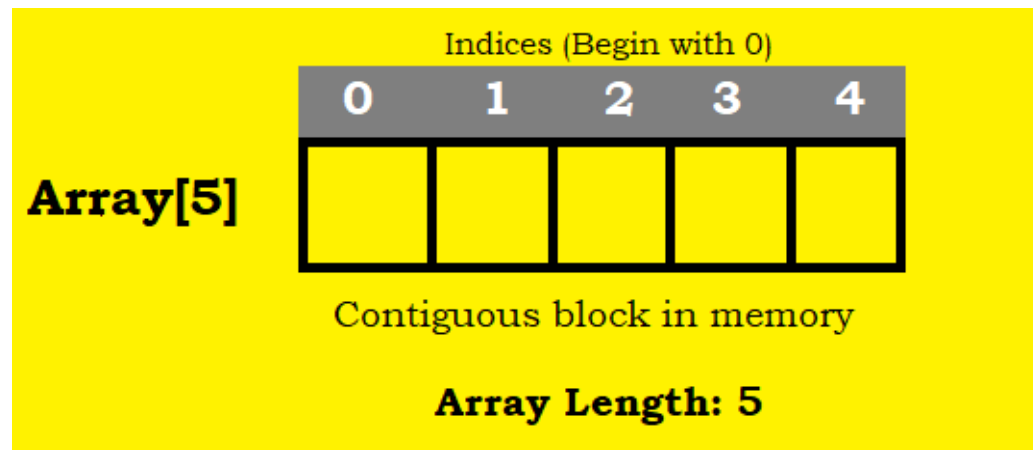
Use Function Inside Another Function

```
//function calling another function
function doit(ho) {
  return ho * 2;
}

function langPerHours(a, b) {
  const d = doit(a);
  const e = doit(b);
  return `${d} ${e}`;
}
console.log(langPerHours(1, 1));
```

Array

In JavaScript, an array is a data structure that allows you to store a collection of values, including other arrays or objects, in a single variable. To create an array, you can use the array literal notation, which consists of enclosing a comma-separated list of values in square brackets:



```
//arrays = array literal and array constructor syntax
const langs = ["javascript", "python", "html"];
console.log(langs);

const newlangs = new Array("javascript", "python", "html");
console.log(newlangs);

console.log(langs[0]);
console.log(langs[1]);
console.log(langs[2]);

//length of array
console.log(langs.length);
console.log(langs[langs.length - 1]);

//change value of element of array
langs[2] = "java";
console.log(langs);

//one array inside another array (2 dimensional array)
const tryAnother = ["javascript", 0, true, langs];
console.log(tryAnother);
```

Array With Const keyword

it's a good practice to create array and object with const keyword.

In JavaScript, the const keyword is also used to declare variables that cannot be reassigned. However, it does not make an array read-only in the same way as in C or C++. When an array is declared with the const keyword in JavaScript, the variable itself is read-only, but the contents of the array can still be modified.

```
const array = [2, 4, 6, 8, 10];  
array = [3, 6, 9, 12];    //error  
  
let array = [2, 4, 6, 8, 10];  
array = [3, 6, 9, 12];    //no error
```

Loop An Array Element Forward

```
const array = [2, 4, 6, 8, 10];  
  
for (let i = 0; i < array.length; i++) {  
    console.log(array[i]);  
}
```

Loop An Array Element Backward

```
const array = [2, 4, 6, 8, 10];  
  
for (let i = array.length - 1; i >= 0; i--) {  
    console.log(array[i]);  
}
```


Array Methods

There are many other methods and properties available for working with arrays in JavaScript, such as `slice()`, `splice()`, `concat()`, `join()`, `length`, and more.

```
//array methods  
const lang = ["js", "python"];  
lang.push("java");  
console.log(lang);  
  
lang.unshift("c++");  
console.log(lang);  
  
lang.pop();  
console.log(lang);  
  
lang.shift();  
console.log(lang);  
  
console.log(lang.indexOf("js"));  
console.log(lang.includes("js"));
```

Object

In JavaScript, an object is an unordered collection of key-value pairs.

Each key-value pair is called a property. The key of a property can be a string. And the value of a property can be any value, e.g., a string, a number, an array, and even a function. JavaScript provides you with many ways to create an object. The most commonly used one is to use the object literal notation.

```
//object doesn't have order or index
const myObject = {
  key1: "value1",
  key2: "value2",
  key3: "value3",
};
```

```
//objects
const shivang = {
  fullName: "shivang",
  job: "front-end",
  language: "javascript",
};
console.log(shivang);

//access object properties with dot or bracket
console.log(shivang.fullName);
console.log(shivang["fullName"]);

//modifying properties
shivang.fullName = "gunjan";

delete shivang.job;

//Checking if a property exists
console.log("fullName" in shivang);

//add new properties to objects
shivang.newProperty = "new";
console.log(shivang);

shivang["anotherNew"] = "new";
console.log(shivang);
```

Method in Object

A Object methods are functions that are stored as object properties. They can be accessed and called in the same way as object properties

```
//function in object as value
const shivang = {
  name: "shivang",
  job: "front-end",
  language: "javascript",
  AdmissionYear: 2019,
  calGraduate: function (year) {
    return shivang.AdmissionYear + 4;
  },
};
```

This keyword

In JavaScript, 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**.

```
const shivang = {
  name: "shivang",
  job: "front-end",
  language: "javascript",
  fullInfo: function () {
    console.log(`this is ${this.name}`);
  },
};
shivang.fullInfo();
```

String and String Method

JavaScript strings are for storing and manipulating text. In JavaScript, a string is a sequence of characters used to represent text. Strings are one of the primitive data types in JavaScript, which means they are not objects but rather a simple value that can be assigned to a variable.

```
const str = "i am string";
const str1 = 'i am string';
const str2 = `i am string`;
```

```
const firstName = "Gamer";
const lastName = "Blame";
const city = "bareilly";
const fullName = firstName + " " + lastName;

console.log(firstName);
console.log(firstName.length);
console.log(firstName === lastName);
console.log(firstName[0]);
console.log(firstName[3]);
console.log(firstName[firstName.length - 1]);
console.log(firstName === lastName);

//string methods
firstName.toUpperCase; //captial
firstName.toLowerCase; //small
firstName.concat(lastName, city); //add strings
firstName.slice(1, 4); //create new string
firstName.split(" "); //divide string
firstName.trim(); //remove whitespace
firstName.indexOf("g"); //true or false
firstName.includes("r"); //index of substring
```

Math Object

The JavaScript Math object allows you to perform mathematical tasks on numbers. The Math object provides mathematical constants and functions. Some common functions include Math.round(), Math.random(), and Math.max().

```
const PI = 22.7;
```

javascript provides us many math object so that we don't have to store them in variable , use math object directly

```
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 object methods
Math.round(2.5);  //Returns rounded to its nearest integer
Math.ceil(2.2);   //Returns rounded up to its nearest integer
Math.floor(2.6);  //Returns rounded down to its nearest integer
Math.trunc(5.3);  //Returns the integer part of x (new in ES6)
Math.abs(-4.7);   //returns the absolute (positive) value
Math.sign(-4);    //1 if the number is positive,-1 negative, 0
zero
Math.pow(8, 2);   //returns the value of x to the power of y
Math.min(0, 150); //find the lowest value
Math.max(0, 150); //find the highest value
```

Math Random

In JavaScript, the `Math.random()` method is used to generate a random number between 0 and 1. `Math.random()` returns a random number between 0 (inclusive), and 1 (exclusive).

```
//Returns a random number:  
Math.random();  
  
//Returns a random integer from 0 to 9:  
Math.floor(Math.random() * 10);  
  
//Returns a random integer from 0 to 10:  
Math.floor(Math.random() * 11);  
  
//Returns a random integer from 0 to 99:  
Math.floor(Math.random() * 100);  
  
//Returns a random integer from 1 to 10:  
Math.floor(Math.random() * 10) + 1;
```

Generate OTP Using Math.random

```
let one = Math.floor(Math.random() * 10);  
let two = Math.floor(Math.random() * 10);  
let three = Math.floor(Math.random() * 10);  
let four = Math.floor(Math.random() * 10);  
  
let otp = `${one} ${two} ${three} ${four}`;  
console.log(`your otp is ${otp} for 10 minutes`);
```

Date Object

```
//create a date
const now = new Date();
console.log(now);

const now = new Date("dec 22 2023 18:22:02");
console.log(now);

//month are zero based like array (jan = 0 , feb = 1 , ...)
//days are also zero based (sunday = 0 , monday = 1 , ...)
const now = new Date(2023, 0, 12, 13, 14, 15);
console.log(now);

//javascript date are started from 1970
const now = new Date(0);
console.log(now);

//date are object so we can use methods
const now = new Date();
console.log(now);

console.log(now.getDate());
console.log(now.getDay());
console.log(now.getHours());
console.log(now.getMinutes());
console.log(now.getMonth());
console.log(now.getFullYear());

//set date and time and other
now.setDate(2);
now.setFullYear(2030);
now.setMonth(2);
```

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:

```
for (let i of array) {  
  //body of loop  
}
```

```
let array = [2, 4, 6, 8, 10];  
  
for (let i of array) {  
  console.log(i);  
}
```

For In Loop

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

```
let myObject = {  
  first: "one",  
  second: "two",  
  third: "three",  
};  
  
for (let i in myObject) {  
  console.log(myObject[i]); //replace myObject[i] with i  
}
```

Break and Continue

The break statement is used to exit a loop early, regardless of whether the loop condition is still true or not. The continue statement is used to skip the current iteration of a loop and move on to the next one.

Some Object Method

In JavaScript, `Object.entries()`, `Object.keys()`, and `Object.values()` are methods that allow you to extract data from an object in different ways. These methods are useful for iterating over the properties of an object, or for extracting specific data from an object.

`Object.entries()` returns an array of key-value pairs for a given object. Each key-value pair is represented as an array where the first element is the key and the second element is the value.

`Object.keys()` returns an array of the keys in a given object.

`Object.values()` returns an array of the values in a given object.

//`Object.keys(obj)` return an array of object property name

//`Object.values(obj)` return array of object values

//`Object.entries(obj)` return an array of [key,value] pair

```
//Object.keys(obj) return an array of object property name
//Object.values(obj) return array of object values
//Object.entries(obj) return an array of [key,value] pair
```

```
//Object.entries
const array = [2, 4, 6, 8, 10, 12, 14, 16, 20];
```

```
let objMethod = Object.entries(array);
```

```
//normal for of loop
for(let i of array){
    console.log(i);
}
```

```
//loop with index number
for (let [i, j] of objMethod) {
    console.log(i, j);
}
```

```
//Object.keys and object.values
const person = {
  fullName: "blame",
  email: "blame@gmail.com",
  education: "btech",
  branch: "information Technology",
  city: "noida",
  job: "front-end",
  Experience: 1,
  fullStack: true,
};

let objectKeys = Object.keys(person);
let objectValues = Object.values(person);

console.log(objectKeys , objectValues);

for(let i of objectKeys){
  console.log(i);
}

for (let j of objectValues) {
  console.log(j);
}
```

Destructuring

In JavaScript, destructuring is a way to extract values from arrays and objects and assign them to variables.

Array Destructing

Array destructuring is a feature in JavaScript that allows you to extract elements from an array and assign them to variables in a more concise and readable way.

There are a few important things to note about array destructuring:

- 1.The variables being assigned to must be enclosed in square brackets ([]).
- 2.The number of variables being assigned to must match the length of the array being destructured.
- 3.The order in which the variables are declared corresponds to the order in which the values are extracted from the array.
- 4.You can skip values in the array by leaving out variables in the destructuring assignment.
- 5.You can also assign default values to variables in case the corresponding value in the array is undefined.

```
//array destructing
const languageWeb = ["html", "css", "js", "python"];

// old way or before destructing
let a = languageWeb[0];
let b = languageWeb[1];
let c = languageWeb[2];
let d = languageWeb[3];

// after destructing
let [a, b, c, d] = languageWeb;

//escape elements
let [a, , , d] = languageWeb;

//default values for variable
let [a, , , d, e = 20] = languageWeb;

//swap two variable using destructing
let a = 10;
let b = 20;
[a, b] = [b, a];
```

Object Destructuring

Object destructuring is a feature in JavaScript that allows you to extract properties from an object and assign them to variables in a more concise and readable way. It allows you to unpack properties from objects into distinct variables.

There are a few important things to note about object destructuring:

1. The variables being assigned to must have the same name as the properties being destructured.
2. You can use a different variable name by providing an alias using the syntax `propertyName: variableName`.
3. You can assign default values to variables in case the corresponding property in the object is undefined.

In this syntax:

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

```
//object destructing
const mylang = {
  web: "javascript",
  andriod: "java",
  backend: "nodejs&django",
  ai: "python",
};

//old way or before destructing
let a = mylang.web;
let b = mylang.andriod;
let c = mylang.backend;
let d = mylang.ai;

//after destructing
//syntax
// let { property1: variable1, property2: variable2 } = object;

let { web: a, andriod: b, backend: c, ai: d } = mylang;

//default values
let { web: a = 10, andriod: b = 20, backend: c = 40, ai: d } = mylang;

//same variable with same property name
({ web, andriod, backend, ai } = mylang);
//The code is enclosed in parentheses () to avoid any potential
syntax errors due to the fact that the curly braces {} are also
used to define code blocks in JavaScript.
```

Spread Operator

The spread operator (...) is used to spread the elements of an array or the properties of an object into another array or object. The spread operator in JavaScript is a powerful feature that allows developers to easily spread or expand the contents of an iterable (such as an array, string, or object) into another data structure. It is denoted by three dots (...).

```
// spread operator (right hand side of =)
// spread operator in arrays

// random arrays
const one = [1, 2, 3, 4, 5];
const two = [6, 7, 8, 9, 10];

//log spread arrays
console.log(...one);
console.log(...two);

//copy array using spread
const three = [...one];
console.log(three);

//store them in variable
const oneTwo = [...one, ...two];
const oneTwo2 = [10, 20, 30, ...one, ...two];
console.log(oneTwo);
console.log(oneTwo2);

//spread on string
const mess = "message";
console.log(...mess);

// input in array then use spread to send it as separate in
function as parameters
function addThree(a, b, c) {
  return a + b + c;
}
const arr = [1, 2, 3];
console.log(addThree(...arr));
```

```
// spread operator in object
// While the spread operator (...) is primarily used with arrays
in JavaScript, it can also be used with objects in some
situations. However, objects are not iterable by default, so
using the spread operator directly on an object will not work.
```

```
const myObject1 = {
  first: "india",
  second: "pakistan",
};
```

```
//TypeError: obj is not iterable
console.log(...myObject1);
```

```
//use spread object inside {}
console.log({ ...myObject1 });
```

```
const myObject2 = {
  third: "russia",
  fourth: "usa",
};
console.log({ ...myObject1 });
```

```
//adding object using spread
const addObject = { ...myObject1, ...myObject2 };
const addObject3 = { ...myObject1, ...myObject2, apple: "no" };
console.log(addObject);
console.log(addObject3);
```

```
//copy object using spread
const myObject = { ...myObject1 };
console.log(myObject);
```

Rest Operator

The rest operator (...) is used to capture the remaining elements of an array or the remaining properties of an object into a new array or object.

When the spread operator is used as a parameter, it is known as the rest parameter.

```
// rest operator (left hand side of =)
const array = [4, 6, 8, 10];
const arrayTwo = [12, 14, 16, 18];
const [a, b, ...ending] = array;
console.log(a, b, ending);

//kon rest kon spread Btaao
// right hand side = spread
// left hand side = rest
const [x, y, z, ...remain] = [...array, ...arrayTwo];

//rest parameter
// When the spread operator is used as a parameter in function
it is known as the rest parameter
function sum(...num) {
  let total = 0;
  for (let number of num) {
    total = total + number;
  }
  return total;
}
console.log(sum(1, 4, 5, 6));

//Rest in object
const person = {
  fullName: "blame",
  city: "ncr",
  country: "india",
  occupation: "Developer",
};

//() are not required when using rest in object
const { fullName, city, ...details } = person;

console.log(fullName, city, details);
```

Sets

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

In JavaScript, a set is an object that stores a collection of unique values of any type. The values can be primitive types such as numbers, strings, or booleans, as well as object references.

While Set is technically an object in JavaScript, it is often treated as a separate data structure or collection type because of its unique behavior and functionality.

Important points about sets:

1. Sets only allow unique values. If you try to add a value that's already in the set, it will be ignored
2. Sets are not indexed. This means that you can't access values in a set by their position, like you can with an array.
3. Sets can contain values of any type, including objects.
4. Sets can be converted to arrays using the spread operator:

```
//sets

//create empty set
const mySet = new Set();

//create a set
let setObject = new Set([1, 2, 3, 4]);
let setObject1 = new Set([1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 9, 1]);
let setObject2 = new Set(["one", "two", "three", "four"]);
let setObject3 = new Set([true, false, true, false]);
let setObject4 = new Set([1, 2, "three", true, { firstName: "blame" }]);

//type of set = object
console.log(typeof mySet);

//convert set into array without duplicate values
let setObject = new Set([1, 2, 3, 4]);
console.log([...setObject]);
```



```
//sets methods
const myLang = new Set([]);
myLang.add(1); //add new element at last of set
myLang.add(2);
myLang.add(3);
myLang.add(4);

console.log(myLang.size); //size is number of element in set

myLang.has(4); //has return true or false
myLang.has(6);

myLang.delete(2); //delete removes the particular element in set
myLang.delete(4);

myLang.clear(); //delete all element in set

//set is also a object so we can use loop
for (let i of myLang) {
  console.log(i);
}

//weak sets
const weakSet = new WeakSet([ { first: "one" }, { second: "two" } ]);
```

WeakSets

A WeakSet is similar to a Set except that it contains only objects. Since objects in a WeakSet may be automatically garbage-collected, a WeakSet does not have size property. Like a WeakMap, you cannot iterate elements of a WeakSet, therefore, you will find that WeakSet is rarely used in practice. In fact, you only use a WeakSet to check if a specified value is in the set.

Maps

A Map holds key-value pairs where the keys can be any datatype. A Map remembers the original insertion order of the keys. A Map has a property that represents the size of the map.

In JavaScript, the Map object is a collection of key-value pairs, where the keys and the values can be of any type. The Map object is similar to an object, but with some important differences.

```
//maps

//create empty map
const myMap = new Map();

//syntax for map
// const myMap = new Map([
//   [key1, value1],
//   [key2, value2],
//   [key3, value3],
// ]);

const myLang = new Map([
  ["first", 200],
  ["second", 500],
  [true, 200],
]);

console.log(typeof myLang); //map are also object

//map methods (most of map methods are similar to sets)
myLang.set(true, 400); //unique are allowed , similar are
ignored
myLang.set(true, 700);
myLang.set(true, 400);

myLang.size;
myLang.get("first");
myLang.delete("second");
myLang.clear();
```

JavaScript Hoisting

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

JavaScript hoisting is a term used to describe the behavior of how variable declarations and function declarations are processed in JavaScript. Hoisting refers to the way that JavaScript interprets code during the compilation phase before executing it. In JavaScript, variable and function declarations are moved to the top of their respective scopes. This means that regardless of where a variable or function is declared in a code block, it is moved to the top of the code block before any other code is executed.

It's important to note that only the declaration of a variable or function is hoisted, not its initialization or assignment. In the case of a variable declaration, the variable is initialized to undefined by default.

```
//using function before defining it is possible because of
Hoisting

console.log(greetEveryone());

function greetEveryone() {
  return "variable and function declarations are moved to the
top";
}

//hoisting works differently for var, let, and const
console.log(a);
var a = 10;      //var = undefined

console.log(b);
let b = 20;      //let and const = reference error

//The reason that hoisting works differently for var, let, and
const variables is because of the way that they are scoped in
JavaScript
```

Reference Vs Value

In JavaScript, variables can hold either primitive values or reference values. Understanding the difference between them is important as it affects how variables are copied and passed around in your code.

Primitive values (e.g. numbers, strings, booleans) are stored directly in memory and have a fixed size. When you assign a primitive value to a variable or pass it as an argument to a function, a copy of the value is created and assigned to the new variable. Any changes made to the new variable will not affect the original value.

Reference values (e.g. objects, arrays, functions) are stored as a reference in memory, rather than the actual value itself. When you assign a reference value to a variable or pass it as an argument to a function, the variable or parameter receives a reference to the original value, rather than a copy of the value. Any changes made to the new variable will affect the original value.

```
// Primitive value example
let x = 5;
let y = x; // y is now a copy of x
y = 10; // changing y does not affect x

console.log(x); // Output: 5
console.log(y); // Output: 10

// Reference value example
let arr1 = [1, 2, 3];
let arr2 = arr1; // arr2 is now a reference to arr1

arr2.push(4); // changing arr2 affects arr1

console.log(arr1); // Output: [1, 2, 3, 4]
console.log(arr2); // Output: [1, 2, 3, 4]
```

Primitive data types are number , string , Boolean , undefined , null , bigint , symbol
primitive use stack memory and you get copy of that variable

Non-primitive data types are object , array , date
non-primitive use heap memory and you get reference of that object

Javascript Best practices

1. Declarations on Top

It is a good coding practice to put all declarations at the top of each script or function. This will Give cleaner code Provide a single place to look for local variables Make it easier to avoid unwanted (implied) global variables Reduce the possibility of unwanted re-declarations

2. Initialize Variables

It is a good coding practice to initialize variables when you declare them. This will: Give cleaner code Provide a single place to initialize variables Avoid undefined values

3. Declare Objects and arrays with const keyword

Declaring objects with const will prevent any accidental change of type and Declaring arrays with const will prevent any accidental change of type

4. Don't Use new Object()

Use "" instead of new String() , Use 0 instead of new Number()

Use false instead of new Boolean() , Use {} instead of new Object()

Use [] instead of new Array() , Use /()/ instead of new RegExp()

Use function () {} instead of new Function()

5. Beware of Automatic Type Conversions

JavaScript is loosely typed. A variable can contain all data types. A variable can change its data type

6. Use === Comparison

The == comparison operator always converts (to matching types) before comparison. The === operator forces comparison of values and type

7. Use Parameter Defaults

If a function is called with a missing argument, the value of the missing argument is set to undefined. Undefined values can break your code. It is a good habit to assign default values to arguments.

8. End Your Switches with Defaults

Always end your switch statements with a default. Even if you think there is no need for it.

JS Best Practices : <https://code.tutsplus.com/tutorials/24-javascript-best-practices-for-beginners--net-5399>

Warning :

Use of formatter and linter after you code in JS for sometimes so that you have learn the syntax , without learning syntax it is not recommended to use these formatter and linters

Formatter And linter

A **formatter** is a tool that automatically adjusts the formatting of your code to follow a specific set of rules or guidelines. It can be used to ensure consistency in code style across a project or team, and to make code more readable and maintainable.

A **linter**, on the other hand, is a tool that analyzes your code for potential errors, bugs, or stylistic issues, and provides feedback or warnings to help you improve your code quality. It can help catch common mistakes, such as syntax errors or unused variables, and can also enforce coding standards and best practices.

Both formatters and linters are commonly used in software development to improve code quality, reduce errors, and make the code more readable and maintainable. VSCode has built-in support for various formatters and linters for different programming languages, and you can also install additional extensions for more advanced formatting and linting capabilities.

Use Of Formatter

Use prettier with format on save in vscode

- 1.Install prettier which is on vscode extension
- 2.In vscode setting setup the formatter as prettier esbenp prettier vscode
- 3.Search for format on save and then check it
- 4.Use prettier every time you save document

Use Of linter

Use eslint in vscode

- 1.Install eslint in vscode from extension
- 2.copy paste this "npm i -d eslint" in terminal and enter
3. Now search npm test lint Airbnb and use npm 5+ command

<https://github.com/airbnb/javascript#objects> : Read Air BNB Style guide

NPM – (Node Package Manager)

npm stands for Node Package Manager. It is a command-line interface tool that allows developers to install, manage, and share reusable code packages written in JavaScript.

npm is typically used with Node.js, a JavaScript runtime environment that allows developers to run JavaScript code on the server-side. Node.js comes with npm pre-installed, which makes it easy to manage dependencies and packages for your Node.js projects.

With npm, you can easily install and manage packages that are available in the npm registry, which is a large public repository of over a million packages. You can also create your own packages and publish them to the npm registry, making them available for others to use.

npm is an essential tool for modern web development, allowing developers to easily manage dependencies and share code with others in the community.

Install NodeJS

NPM Automatically Install with NodeJS

Visit NPM Create your account

To install any package we have to install it locally or globally

Installing Live Server with npm Globally

Use ctrl+backtick to open terminal

npm install -g live-server

-g is used for global package

Use cls to clear terminal

if live-server is install globally

Use **live-server** to run server and ctrl+C to terminate server

if live server is not install globally use this

npm run live-server and ctrl+C to terminate server

Some NPM Commands

Here are some basic npm commands that you can use to manage packages and dependencies. These are just some of the basic npm commands. You can find more commands and options in the npm documentation.

npm init : Initializes a new npm package in the current directory and creates a package.json file.

npm install : Installs all the dependencies specified in the package.json file.

npm install <package-name> : Installs a specific package and adds it to the dependencies section of the package.json file.

npm install --save-dev <package-name> : Installs a package as a dev dependency and adds it to the devDependencies section of the package.json file.

npm uninstall <package-name> : Uninstalls a package and removes it from the dependencies or devDependencies section of the package.json file.

npm update : Updates all packages to their latest versions.

npm outdated : Lists all the outdated packages in your project.

npm search <package-name> : Searches the npm registry for packages with the specified name.

npm ls : Lists all the installed packages and their versions.

npm run <script-name> : Runs a script specified in the scripts section of the package.json file.

npm publish : Publishes a package to the npm registry.

npm login : Logs in to the npm registry.

npm whoami : Displays the username of the currently logged in user.

npm audit : Checks for vulnerabilities in your project's dependencies.

npm ci : Installs dependencies based on the package-lock.json file, which is useful for setting up a clean build environment.

Function Methods or Explicit binding

In JavaScript, call, apply, and bind are three methods that can be used to manipulate the this keyword in a function.

Call method

Apply method

Bind method

Call Method

In JavaScript, the call() method is a built-in function that allows you to call a function with a specified this value and arguments provided individually.

```
//call method

const person = {
  fullName: "blame",
  job: "full Stack Developer",
  experience: 1,
  fullInfo: function () {
    return `my name is ${this.fullName} and i am ${this.job}
having ${this.experience} years experience`;
  },
};

const person2 = {
  fullName: "element",
  job: "backend developer",
  experience: 4,
};

//using call method
const result = person.fullInfo.call(person2);
console.log(result);

//this keyword in person is still points to person
console.log(person.fullInfo());
```

Apply Method

In JavaScript, the `apply()` method is similar to the `call()` method in that it allows you to call a function with a specified `this` value and arguments. However, instead of passing arguments individually, you pass them as an array.

```
//apply method : only difference between call and apply is how
we pass argument

//apply method
//same as call method , only difference is that in apply we give
arguments in arrays

const person1 = {
  one: 1,
  two: 2,
  add: function (arg1, arg2) {
    return `${this.one} ${this.two} and arg is ${arg1} and
    ${arg2}`;
  },
};

const person2 = {
  one: 300,
  two: 400,
};

const result = person1.add.apply(person2, ["1st arg", "2nd
arg"]);
console.log(result);

//this in person1 is still points to person1
console.log(person1.add("1", "2"));
```

Bind Method

In JavaScript, the `bind()` method is used to create a new function with a specified `this` value and any number of arguments that are pre-specified.

```
//bind method : same as call method only difference is that it
return as function

const person1 = {
  one: 1,
  two: 2,
  add: function (arg1, arg2) {
    return `${this.one} ${this.two} and arg is ${arg1} and
    ${arg2}`;
  },
};

const person2 = {
  one: 300,
  two: 400,
};

const result = person1.add.bind(person2, "0.1", "0.2");
console.log(result); //return as function
const finalResult = result();
console.log(finalResult);
```

Arrow Function Revisit

Arrow functions do not have their own this binding, so they use the this value of the surrounding execution context. This behavior is different from regular functions (function declaration), which have their own this binding.

Arrow function doesn't have this keyword

Arrow function inherit this from parent

```
//this keyword in arrow function points to parent this
const person = {
  fullName: "blame",
  testing: `${this}`,
  info: () => `${this.fullName}`,
};

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

Arguments Object

The arguments object is a built-in JavaScript object that is automatically created inside every function. It contains an array-like list of all the arguments passed to the function when it is called. Although you can access its elements using an index like an array, the arguments object does not have all the array methods like push, pop, and slice. However, you can convert the arguments object to a proper array using the Array.from() method or by using the spread operator (...).

```
const result = function (a, b, c) {
  console.log(arguments); //array like object
  console.log(arguments[0]); //access element like array
  console.log([...arguments]); //convert argument into array
  using spread operator

  //it will be more wise if we use rest parameter
};

result(2, 4, 6);
```

Error Handling in Javascript

Try...Catch...Finally

The try statement defines a code block to run (to try).

The catch statement defines a code block to handle any error.

The finally statement defines a code block to run regardless of the result.

The throw statement defines a custom error.

```
//try and catch
console.log("gg");
console.log(gg);
console.log("op");
//Result : you will get error becoz gg is not defined and your
script will stop at midway
```

```
//try and catch
console.log("gg");
console.log(ddlj);
console.log("dd");

try {
  console.log(gg);
} catch (error) {
  console.log("there is a error", error);
} finally {
  console.log("finally we are here");
}

console.log("op");

let a = 10;
if(a==10) throw new Error("you did mistake");
```

JSON

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa).

JSON is a format for storing and transporting data. JSON is often used when data is sent from a server to a web page.

What is JSON? JSON stands for JavaScript Object Notation. JSON is a lightweight data interchange format. JSON is language independent. JSON is "self-describing" and easy to understand. The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

To Understand Better : take example of **Weather API**

XML VS JSON

XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

JSON

```
{ "empinfo" :
  {
    "employees" : [
      {
        "name" : "James Kirk",
        "age" : 40,
      },
      {
        "name" : "Jean-Luc Picard",
        "age" : 45,
      },
      {
        "name" : "Wesley Crusher",
        "age" : 27,
      }
    ]
  }
}
```

JSON Syntax Rules

Data is in name/value pairs where name should be enclosed in double quotes

Data is separated by commas

Curly braces hold objects

Square brackets hold arrays

single quotes are not allowed

comment and undefined is not allow

data.json

how to write json file example and compare it with javascript object

```
{
  "fullName": "blame Gamer",
  "highestScore": 233,
  "isGamer": true,
  "address": null,
  "favouriteGamer": ["gta 5", "witcher 3", "csgo", "takken 3"],
  "experience": [
    {
      "companyName": "xyz software technology",
      "duration": 2
    },
    {
      "companyName": "abc private LTM",
      "duration": 5
    }
  ]
}
```

JSON Two Method

parse : parse convert json file into javascript object

stringify : convert JavaScript object or value into JSON string

```
const myJson = `{
  "fullName": "blame Gamer",
  "highestScore": 233,
  "isGamer": true,
  "address": null,
  "favouriteGamer": ["gta 5", "witcher 3", "csgo", "takken 3"],
  "experience": [
    {
      "companyName": "xyz software technology",
      "duration": 2
    },
    {
      "companyName": "abc private LTM",
      "duration": 5
    }
  ]
}`;

//parse a JSON string and convert it into a JavaScript object
const jsonAsObject = JSON.parse(myJson);
console.log(jsonAsObject);

//convert JavaScript object or value into JSON string
const jsonAsString = JSON.stringify(jsonAsObject);
console.log(jsonAsString);
```


Advance Functions Concept

Closer Look At Functions

Default Parameter

In JavaScript, default parameters are used to set default values for function parameters when no value or undefined is passed as an argument. ES6 allows function parameters to have default values.

When the function is called with no arguments or undefined values for those parameters, the default values are used. If values are passed as arguments, the passed values will override the default values.

```
//if function is called without argument , parameter will be undefined
```

```
function greetTwo(a, b) {  
  console.log(`${a} ${b} have a good day`);  
}
```

```
greetTwo();
```

```
//default parameter
```

```
function greet(message = "Hi") {  
  console.log(`${message} have a good day`);  
}
```

```
greet(); //no argument
```

```
greet("Hello"); // argument
```

Higher Order Function

In JavaScript, a higher-order function is a function that takes one or more functions as arguments, and/or returns a function as its result. In other words, a higher-order function operates on other functions, either by taking them as arguments or returning them as values.

If one of them is true it is higher order function

1. functions as arguments
2. Returns a function as result

Functions Returning Functions

```
//function with no arguments
```

```
function hello() {  
  return function () {  
    console.log("hello");  
  };  
}  
  
const result = hello();  
result();
```

```
//function with arguments
```

```
function sayHello(firstName) {  
  
  function hello(lastName) {  
    console.log(`hello ${firstName} ${lastName}`);  
  }  
  
  return hello;  
}
```

```
const result = sayHello("blame");  
result("gamer");
```

Function as Argument

```
//function as argument  
  
function fullName(firstName, greetPlace) {  
  const greet = greetPlace(firstName);  
  console.log(greet);  
}  
  
function sayHello(firstName) {  
  return `hello ${firstName}`;  
}  
  
fullName("blame", sayHello);
```

First Class Function

In JavaScript, functions are considered **first-class citizens**, which means that they can be treated like any other value in the language, such as numbers, strings, or objects. This allows them to be passed as arguments to other functions, returned as values from functions, and assigned to variables or object properties.

Here are some examples of how functions can be used as first-class values in JavaScript:

1. Functions can be assigned to variables
2. Functions can be passed as arguments to other functions
3. Functions can be returned as values from other functions

Map Filter Reduce

map(), **filter()**, and **reduce()** are three powerful array methods in JavaScript that are used to manipulate and transform arrays.

Map Method

map() is used to create a new array by transforming each element of an existing array. It takes a function as an argument that is applied to each element of the array, and returns a new array of the same length where each element is the result of applying the function to the corresponding element of the original array.

```
//map method using arrow function
const twoTimes = [1, 2, 4, 6, 8, 10, 11];

const a = twoTimes.map((i) => i + 2);

console.log(a);
```

```
//map method using anonymous function
const arr = [1, 2, 4, 5, 6, 8, 10];
const result = arr.map(function (element , index , array) {
    const one = element + 1;
    return one;
});

console.log(result);
```

Filter Method

filter() is used to create a new array that contains only the elements of an existing array that pass a certain test. It takes a function as an argument that is applied to each element of the array, and returns a new array that contains only the elements for which the function returns true.

```
//filter method using arrow function
const twoTimes = [1, 2, 4, 6, 8, 10, 11];

const b = twoTimes.filter((i) => i % 2 === 0);
console.log(b);
```

```
//filter method using anonymous function
const arr = [1, 2, 4, 5, 6, 8, 10];
const result = arr.filter(function (element, index, array) {

    // console.log(element);
    // console.log(index);
    // console.log(array);

    return element % 2 == 0;
});

console.log(result);
```

Reduce Method

reduce() is used to apply a function to all the elements of an array and reduce them to a single value. It takes a function as an argument that is applied to each element of the array, and a starting value. The function is applied to the first two elements of the array and the result is used as the starting value for the next application of the function to the next element in the array, and so on, until all the elements of the array have been processed.

```
//reduce method
const twoTimes = [1, 2, 4, 6, 8, 10, 11];

const c = twoTimes.reduce((accumulator, currentValue) =>
  accumulator + currentValue);

console.log(c);
```

Immediately Invoked Function Expressions (IIFE)

IIFE stands for Immediately Invoked Function Expression. It is a way to define and immediately call a function in JavaScript. A JavaScript immediately invoked function expression is a function defined as an expression and executed immediately after creation. The following shows the syntax of defining an immediately invoked function expression:

```
//syntax of IIFE
// () ();
(function () {
  //body of function
})();
```

```
//IIFE with Anonymous function
(function () {
    console.log("you didn't call me i am self invoke");
})();

//IIFE with arrow function
(() => console.log("you didn't call me i am self invoke"))();

//IIFE with function declaration
(function myFunction() {
    console.log("you didn't call me i am self invoke");
})();
```

Callbacks functions

In JavaScript, a callback function is a function that is passed as an argument to another function and is executed when that function has completed its task. The purpose of a callback function is to allow asynchronous programming and to execute code when a particular event happens.

```
//callback functions
function add(a, b) {
    return a + b;
}

function minus(a, b) {
    return a - b;
}

function calculate(x, y, operation) {
    return operation(x, y);
}

calculate(10, 45, add);
```

Closure & Lexical Scoping

Lexical scoping is a way for JavaScript to determine the scope of a variable based on where it is declared in the code. In other words, it refers to the visibility and accessibility of variables within a particular context of code.

Closure is a fundamental concept in JavaScript that refers to the ability of a function to access variables from its outer (enclosing) function, even after the outer function has returned. When a function is defined inside another function, the inner function forms a closure with the outer function's variables. This means that the inner function has access to the outer function's variables, even after the outer function has completed execution.

```
//understanding closure with example
function outerFunction() {

    const message = "Hello, world!";

    function innerFunction() {
        console.log(message);
    }

    return innerFunction;
}

const closure = outerFunction();
closure();
```

Summary of Closure

SUMMARY : Closure is "when you have a function defined inside of another function, that inner function has access to the variables and scope of the outer function even if the outer function finishes executing and those variables are no longer accessible outside of that function."

Some Important Array Methods

Reverse method

The **reverse()** method reverses the order of the elements in an array. The reverse() method overwrites the original array.

```
//Reverse method
twotimes = [10, 20, 30, 40, 50, 60, 70, 80];
const reverse = twotimes.reverse();

console.log(reverse);
```

Flat method

The **flat()** method is used to flatten a nested array. It returns a new array that is a flattened version of the original array, with all sub-arrays concatenated into it.

```
//flat method
const arr = [2, 4, [3, 9], [4, 8]];
const result = arr.flat();
console.log(result);

const arr = [2, 4, [3, 9], [4, 8], [1, [2, 4]]];
const result = arr.flat(2);
console.log(result);
```

Every method

The **every()** method is a built-in method in JavaScript that is used to check if all the elements of an array pass a test (specified by a function). It returns a Boolean value, true if all elements pass the test, otherwise false.

```
//Every method
const arr = [1, 2, 4, 5, 6, 8, 10];
const result = arr.every(function (element) {
  return element > 5;
});

console.log(result);
```

Find Method

The **find()** method is a built-in method in JavaScript arrays that is used to find the first element in an array that satisfies a given condition.

```
//find method
const twoTimes = [1, 2, 4, 6, 8, 7, 10];
const result = twoTimes.find((element) => element > 2);
console.log(result);
```

findIndex Method

The **findIndex()** method executes a function for each array element. The **findIndex()** method returns the index (position) of the first element that passes a test. The **findIndex()** method returns -1 if no match is found. The **findIndex()** method does not execute the function for empty array elements. The **findIndex()** method does not change the original array.

```
//findIndex method
const twoTimes = [1, 2, 4, 6, 8, 7, 10];
const result = twoTimes.findIndex((element) => element > 6);
console.log(result);
```

Some method

The **some()** method checks if any array elements pass a test (provided as a callback function). The **some()** method executes the callback function once for each array element. The **some()** method returns true (and stops) if the function returns true for one of the array elements. The **some()** method returns false if the function returns false for all of the array elements. The **some()** method does not execute the function for empty array elements. The **some()** method does not change the original array.

```
//some method
const twoTimes = [1, 2, 4, 6, 8, 7, 10];
const result = twoTimes.some((element) => element > 4);
console.log(result);
```

Fill method

The **fill()** method fills specified elements in an array with a value. The fill() method overwrites the original array. Start and end position can be specified. If not, all elements will be filled.

```
//fill method : value start end
const twoTimes = [1, 2, 4, 6, 8, 7, 10];
const result = twoTimes.fill(10, 2, 4);
console.log(result);
```

Sort method

The **sort()** sorts the elements of an array. The sort() overwrites the original array. The sort() sorts the elements as strings in alphabetical and ascending order.

```
//sort method
//ascending order
const twoTimes = [1, 2, 4, 6, 8, 7, 10];
const result = twoTimes.sort((a, b) => a - b);
console.log(result);

//descending order
const result2 = twoTimes.sort((a, b) => b - a);
console.log(result2);
```

If the result is negative, a is sorted before b.

If the result is positive, b is sorted before a.

If the result is 0, nothing changes.

Foreach method

In JavaScript, the **forEach()** method is used to loop over the elements of an array and execute a callback function for each element. The syntax for **forEach()** is as follows:

```
array.forEach(callback(currentValue, index, array) {  
  // code to execute for each element  
});
```

Technically foreach is not a loop but it acts like a loop , foreach is a array method

forEach is not actually a loop in JavaScript, but rather a method that can be used to iterate over the elements of an array and execute a callback function on each element. While it may seem like a loop construct, it is actually a built-in method of the Array object in JavaScript. The **forEach** method is a higher-order function that takes a function as an argument, and this function is executed for each element of the array.

```
const array = [2, 4, 6, 8, 10];  
  
array.forEach(function (element, index, array) {  
  console.log(index, element, array);  
});
```

Important Point

Write Some Function from above Arrow Function

Remember we can pass function in argument as well

Scheduling `setTimeout` and `setInterval`

We may decide to execute a function not right now, but at a certain time later. That's called "scheduling a call". There are two methods for it:

`setTimeout` allows us to run a function once after the interval of time.

`setInterval` allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

Javascript `setTimeout`

`setTimeout` is a built-in JavaScript function that allows you to execute a specified function or code block once after a specified delay in milliseconds.

```
//syntax of setTimeout
setTimeout(function, delay);

// function or any code of block
//delay should be in milliseconds
// like 3000 which is 3 second
//1 second = 1000 millisecond
```

```
let fullName = "blame";

setTimeout(function () {
  console.log(`hello mr ${fullName}`);
  console.log(`how are you today?`);
  console.log(`have a great day`);
}, 3000);
```

Fun activity

```
setTimeout(function () {
  console.log('loading your system files');
}, 4000);

setTimeout(function () {
  console.log('===== 0% completed =====');
}, 8000);

setTimeout(function () {
  console.log('===== 50% completed =====');
}, 12000);

setTimeout(function () {
  console.log('===== 100% completed =====');
}, 15000);

setTimeout(function () {
  console.log('your files are ready...!');
}, 18000);
```

Arguments in setTimeout

```
setTimeout(
  function (myName) {
    console.log('hello my name is ${myName}');
  }, 4000, "blame"
);
```

Cancelling with clearTimeout

A call to `setTimeout` returns a “timer identifier” `timerId` that we can use to cancel the execution.

```
let timer = setTimeout(  
  function (firstName) {  
    console.log(`hello ${firstName}`);  
  }, 4000, "blame"  
);  
  
//log timerid  
console.log(timer);  
  
//This will cancel the timer and prevent the function from  
executing  
clearTimeout(timer);
```

Pass Function in SetTimeout

That doesn't work, because `setTimeout` expects a reference to a function. And here `greet()` runs the function, and the result of its execution is passed to `setTimeout`.

```
const greet = function (fullName) {  
  console.log(`hello there ${fullName}`);  
};  
  
//wrong  
setTimeout(greet(), 4000, "blame");  
  
//right way  
setTimeout(greet, 4000, "blame");
```


Javascript setInterval

The **setInterval()** method calls a function at specified intervals (in milliseconds). The setInterval() method continues calling the function until clearInterval() is called, or the window is closed.

```
//syntax
setInterval( function , delay)

setInterval(function () {
  console.log('hello i am set interval');
}, 2000);
```

clearInterval

In JavaScript, the **clearInterval()** method is used to stop the execution of a function that was previously scheduled to run repeatedly with the setInterval() method.

```
const timer = setInterval(function () {
  console.log("i am time interval");
}, 3000);

clearInterval(timer);
```

parseInt() and parseFloat() and Number()

You can also use the Number() constructor to convert a string to a number in JavaScript. The main difference between parseInt()/parseFloat() and Number() is in how they handle non-numeric characters in the input string.

parseInt() and parseFloat() are designed to extract a numeric value from a string that may contain non-numeric characters. For example, parseInt("123abc") would return the number 123, since it extracts the numeric characters at the beginning of the string and ignores the rest. parseFloat() works similarly, but allows for decimal points and exponents.

Number(), on the other hand, requires that the entire string be a valid number in order to convert it to a number. If the input string contains non-numeric characters, Number() will return NaN, which stands for "not a number".

In most cases, either parseInt()/parseFloat() or Number() will work for converting a string to a number in JavaScript. However, if you need to extract a numeric value from a string that may contain non-numeric characters, parseInt() or parseFloat() may be a better choice.

```
//parseInt use to get integer from string
parseInt("12abc");           //12
parseInt("12ab3c");          //12
parseInt("ab2c");            //NaN
parseInt("12.2");            //12

//parseFloat use to get floating number
//same as parseInt only difference it return decimal numbers
parseFloat("12abc");         //12
parseFloat("12.2abc");       //12.2
parseFloat("12.2abc0.2");    //12.2
parseFloat("abc0.2");        //NaN

//Number() convert into number
Number("21");
Number("21.2425");
Number(true);
Number(false);
```

Web Storage (Local storage and Session storage)

Web storage is a mechanism for storing data on the client-side (i.e., within the user's web browser) that can be accessed and manipulated by JavaScript code running on a web page. It provides a way to store data locally on the user's device without having to send it back to the server, which can improve application performance and reduce network traffic.

There are two types of web storage: `localStorage` and `sessionStorage`.

localStorage is a persistent storage mechanism that allows web applications to store data that will persist even after the browser is closed and reopened. This data is stored in key-value pairs and can be accessed and manipulated by JavaScript code running on the same domain.

sessionStorage, on the other hand, is a temporary storage mechanism that allows web applications to store data for a single browsing session. This data is also stored in key-value pairs and can be accessed and manipulated by JavaScript code running on the same domain. However, when the browser is closed, the data is deleted.

Both localStorage and sessionStorage have a limit on the amount of data they can store, which varies depending on the browser and the device being used. Web storage is commonly used to store user preferences, session information, and other data that needs to be accessed and manipulated by JavaScript code running on the client-side.

Cookies Vs Local Vs Session Storage

	Cookies	Local Storage	Session Storage
Capacity	4kb	10mb	5mb
Browsers	HTML4 / HTML 5	HTML 5	HTML 5
Accessible from	Any window	Any window	Same tab
Expires	Manually set	Never	On tab close
Storage Location	Browser and server	Browser only	Browser only
Sent with requests	Yes	No	No

Local Storage

```
// local Storage
// set local storage
localStorage.setItem("blame", "ProGamer");
localStorage.setItem("blaze", "SSGamer");

//update value using key
localStorage.setItem("blaze", "RRGamer");

//use variable
const fullName = "element";
const game = "30";
localStorage.setItem(fullName, game);

//access local storage items
localStorage.getItem("blame");
localStorage.getItem("location"); //null

//remove from local storage items
localStorage.removeItem("blame");

//remove all data in local storage
localStorage.clear();

//length of local storage
localStorage.length;

//key of local storage
localStorage.key(0); //keys are 0 index based like array
```

```
//using other data types in local storage
const fullName = "User information";
const info = {
  loginTime: "12:00 am",
  duration: "20 minutes",
};

localStorage.setItem(fullName, JSON.stringify(info));

//access
const information = JSON.parse(localStorage.getItem("User information"));
console.log(information);
```

Session Storage

Replace session with local in above code

Most time you will never use Session storage

```
//session storage

//set session storage
const user = "blame";
const info = "gamer";

sessionStorage.setItem(user, info);

//access session storage
sessionStorage.getItem("blame");
```

```
//remove item from session storage
sessionStorage.removeItem("blame");

//clear session storage
sessionStorage.clear();
```

Cookies

Cookies are small pieces of data that are sent from a website to a user's web browser when the user visits the website. These pieces of data are then stored on the user's device and can be retrieved by the website when the user visits it again in the future. Cookies are commonly used to remember user preferences, login information, and other settings. There are two types of cookies: session cookies and persistent cookies. Session cookies are temporary cookies that are erased when the user closes their web browser. Persistent cookies, on the other hand, are stored on the user's device for a longer period of time, often with an expiration date set by the website. Persistent cookies can be used to remember user preferences and settings across multiple sessions. Cookies can also be used to track user behavior and collect information about their browsing habits. This has raised concerns about privacy and data security, and as a result, many web browsers now allow users to control which cookies are stored on their devices and to delete cookies that they no longer wish to keep. It's worth noting that some countries have laws and regulations that require websites to inform users about the use of cookies and obtain their consent before storing them on their devices.

Some browser doesn't allow cookies

go to setting > privacy and security > cookies and other site data > allow all cookies

```
//cookies

//create cookies
document.cookie = "player1 = blame";
document.cookie = "player2 = element";
```

```
//update cookies
document.cookie = "player2 = blaze";

//set expiry date for cookies
document.cookie = "player3 = blaze; expires= Sat,4 Mar 2024
12:00:00 UTC";

//read or see cookies
const result = document.cookie;
console.log(result);
```

Clear cookies

To clear a cookie, you can set its expiration date to a date in the past using the expires attribute. When a cookie's expiration date is in the past, it is considered expired and will be deleted by the browser.

Refactor Code

Refactoring is the process of improving the structure, design, and efficiency of existing code without changing its external behavior. The goal of refactoring is to make code easier to understand, modify, and maintain, while reducing the likelihood of bugs and errors. When you refactor code, you make changes to the internal structure of the code, but the external behavior of the code should remain the same. This means that any tests that were previously passing should still pass after the code has been refactored.

Some common reasons for refactoring code include:

Improving code quality : Refactoring can help eliminate code smells, such as duplicated code, long methods, or unnecessary complexity.

Enhancing performance : Refactoring can help improve the performance of code by optimizing algorithms or reducing the amount of memory used.

Increasing maintainability : Refactoring can make code easier to understand and modify, which can help reduce the cost and effort required to maintain the code over time.

When refactoring code, it's important to have a clear understanding of the existing code and what you want to achieve with the refactoring. It's also important to have a comprehensive suite of tests in place to ensure that the refactored code behaves correctly.

DRY

DRY stands for **"Don't Repeat Yourself"**. It is a software development principle that encourages developers to avoid duplicating code and to keep code as concise and reusable as possible. The goal of DRY is to reduce code complexity, improve maintainability, and reduce the likelihood of bugs or errors.

```
//Normal Code
//sort array in ascending order

const arr = [4, 2, 8, 3, 7, 5, 10];
const result = arr.sort((a, b) => a - b);
console.log(result);

const arr2 = [4, 2, 8, 3, 7, 5, 10];
const result2 = arr.sort((a, b) => a - b);
console.log(result2);

const arr22 = [2, 0, 8, 7, 7, 4, 3, 7, 8, 8];
const result22 = arr.sort((a, b) => a - b);
console.log(result22);

//Refactor Code
function arraySortAscending(arr) {
  const sortArr = arr.sort((a, b) => a - b);
  return sortArr;
}

const result3 = arraySortAscending([2, 140, 5, 4, 45, 65, 45]);
console.log(result3);
```


JavaScript Debuggers

Debugging is the process of testing, finding, and reducing bugs (errors) in computer programs. The first known computer bug was a real bug (an insect) stuck in the electronics.

Debugging is not easy. But fortunately, all modern browsers have a built-in JavaScript debugger. Built-in debuggers can be turned on and off, forcing errors to be reported to the user. With a debugger, you can also set breakpoints (places where code execution can be stopped), and examine variables while the code is executing.

1. Console.log

2. Browser Debugger (using Breakpoint)

```
//using debugger

let a = 10;
let b = 2;
debugger;
for (let i = 1; i <= 10; i++) {
    console.log(i);
}

function add(a, b) {
    return a + b;
}

const result = add(50, 60);
console.log(result);

console.log("1");
console.log("1");
console.log("1");
console.log("1");
```

Modules

JavaScript modules allow you to break up your code into separate files. This makes it easier to maintain a code-base. Modules are imported from external files with the import statement. Modules also rely on **type="module"** in the <script> tag.

In JavaScript, modules are a way of organizing and structuring code into reusable units. Modules allow you to encapsulate related code and expose only the parts that you want to be public, while keeping everything else private. This helps to prevent naming collisions and reduces the risk of unintended consequences.

In the past, JavaScript didn't have a native module system, so developers used various workarounds to achieve module-like functionality, such as immediately-invoked function expressions (IIFEs) or the CommonJS module format. However, in recent years, the ECMAScript specification (the standard that JavaScript is based on) has introduced a native module system.

main.js

```
// import and export

import { greet } from "./another.js";
greet();

import { str as another } from "./another.js";
console.log(another);

//import all
import * as myObject from "./another.js";
myObject.greet();

//using default export
import ps from "./second.js";
console.log(ps);
```

another.js

```
export const greet = () => console.log("another file code");

export const str = "use export so other file can use it";

// export { greet , str };
```

second.js

```
const person = {
  fullName: "blame",
};

export default person;

//export default can be given to only one
//With the export default syntax, you can only export one value
as the default export from a module.
```

Codebase structure will be like :-

Root Folder

--main.js

--another.js

--second.js

Regular Expression or Regex

Regular expressions, also known as regex or regexp, are patterns used to match and manipulate text. They are commonly used in programming languages like JavaScript to validate input, search for patterns in strings, and replace parts of strings.

In JavaScript, regular expressions are represented by the RegExp object. To create a regular expression pattern, you can use the regular expression literal syntax, which encloses the pattern in forward slashes (/pattern/), or you can create a RegExp object with a string that represents the pattern.

Regex in 120 seconds :

<https://www.youtube.com/watch?v=sXQxhojSdZM>

```
const story =
  "Zero is a beautiful number. Not because it is round and cute
  but because what it signifies. It marks the beginning of
  something. It marks the lowest something, or someone can go. It
  is the ultimate foundation to build anything upon.";

//change only first expression
// const result = story.replace("it", "as");
// console.log(result);

//using regex
const regex = /it/g;
const result = story.replace(regex, "as");
console.log(result);
```

To write Regex Easily visit : <https://regexr.com>

Symbol

In JavaScript, **symbols** are a primitive data type that represent unique identifiers. A symbol is created using the `Symbol()` function, which returns a unique symbol value.

we will rarely use symbol in javascript

```
// Number("22") String(22) Boolean(1)
```

```
//Symbol
```

```
const s1 = Symbol("cat");
```

```
const s2 = Symbol(24);
```

```
const s3 = Symbol("cat");
```

```
//every symbol is unique
```

```
console.log(s1 === s3);
```

```
//only use of Symbol is in object
```

```
const myObj = {  
  fullName: "gg",  
  "your name": 24,  
  [s2]: "symbol inside object",  
};
```

```
console.log(myObj["your name"]);
```

```
//wrong way
```

```
myObj.s1 = "gg";
```

```
// right way
```

```
myObj[s1] = "gg";
```

```
myObj[s3] = "gg";
```

Extra Topics

Return in if else

In JavaScript, you can use the return statement within an if-else statement to return a value or terminate the execution of a function based on a condition.

```
function checkNumber(num) {  
  if (num > 0) {  
    return "Positive";  
  } else if (num < 0) {  
    return "Negative";  
  } else {  
    return "Zero";  
  }  
}  
  
checkNumber(5);      // Output: Positive  
checkNumber(-3);     // Output: Negative  
checkNumber(0);      // Output: Zero
```

Another Use Of if-else

```
if (value === true) {  
  console.log("value is truthy");  
} else {  
  console.log("value is falsely");  
}
```

```
if (value) {  
  console.log("Value is truthy");  
} else {  
  console.log("Value is falsy");  
}
```

```
let userLogged = true;

if (!userLogged) console.log("logged");
if (userLogged) console.log("logged");
```

isArray

The `isArray()` method returns true if an object is an array, otherwise false.

```
const arr = [2, 3];
Array.isArray(arr);    //true

const arr = "string";
Array.isArray(arr);    //false
```

isNaN

In JavaScript NaN is short for "Not-a-Number". The `isNaN()` method returns true if a value is NaN. The `isNaN()` method converts the value to a number before testing it.

```
const randomNumber = 22.2;
isNaN(randomNumber);    //false

const randomNumber = "22.2px";
isNaN(randomNumber);    //true
```

toString & toFixed

The `toString()` method returns a date object as a string.

The `toFixed()` method converts a number to a string. The `toFixed()` method rounds the string to a specified number of decimals.

```
//toString : returns string
const now = new Date();
const result = now.getFullYear().toString();

//toFixed
const score = 255.32321;
const scorePerfect = score.toFixed(2);
```

Arrow function Doesn't have Arguments Object

Arrow functions in JavaScript do not have their own arguments object. The arguments object is available within regular functions and provides access to the arguments passed to the function.

Arrow functions, on the other hand, have a different behavior when it comes to handling parameters. They inherit the arguments from their surrounding scope, which means they don't have their own arguments object.

```
//arguments working in normal function
const arrFun = function () {
  console.log(`${arguments}`);
};
arrFun(2, 3);

//error in arrow function
const arrFun = (a, b) => console.log(`${arguments}`);
arrFun(2, 3);
```

Solution : Use Rest Parameter

If you need to access the arguments within an arrow function, you can use the rest parameters syntax (...args) to capture all the arguments into an array.

```
//fix arguments in arrow function
const arrFun = (...arg) => console.log(`${arg} ,
${Array.isArray(arg)}`);
arrFun(2, 3);
```

Declaring Variables Shorthand

```
//longhand
let x;
let y;
let z = 3;

//shorthand
let x,y,z = 3;
```


Nullish coalescing operator (??)

The nullish coalescing (??) operator is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

```
// if (arr === null || arr === undefined) {  
//   console.log("arr is null or undefined");  
// }  
  
//two possible ways  
//if arr is null or undefined  
//if arr have some value  
let arr = 10;  
const result = arr ?? "other than that";  
console.log(result);
```

Optional Chaining (?.)

The optional chaining operator (?.) allows you to access the value of a property located deep within a chain of objects without explicitly checking if each reference in the chain is null or undefined.

```
//optional chaining ?.  
const server = {  
  serverName: "Google",  
  hostCountry: {  
    amercia: "yes",  
    russia: "no",  
  },  
};  
  
const result = server.hostCountry.inida?.serverInfo;
```

Short Circuit

In JavaScript, a "short circuit" refers to the behavior of logical operators when evaluating boolean expressions. JavaScript provides two short-circuiting logical operators: the logical OR (||) and the logical AND (&&).

When using the logical OR operator (||), the evaluation stops as soon as a truthy value is encountered. If the first operand is truthy, the result will be that value, and the second operand won't be evaluated. However, if the first operand is falsy, the result will be the value of the second operand.

Similarly, when using the logical AND operator (&&), the evaluation stops as soon as a falsy value is encountered. If the first operand is falsy, the result will be that value, and the second operand won't be evaluated. However, if the first operand is truthy, the result will be the value of the second operand.

```
//OR stops as soon as a truthy value is encountered
let value1 = true;
let value2 = false;

let result = value1 || value2;
console.log(result);

//AND stops as soon as a falsy value is encountered
let value1 = true;
let value2 = false;

let result = value1 && value2;
console.log(result);
```

Enhanced Object Literals in ES6

Enhanced Object Literals is a feature introduced in ECMAScript 2015 (ES6) that allows for more concise and flexible syntax when defining objects in JavaScript. It provides several enhancements to the traditional object literal syntax.

These enhancements make it more convenient and expressive to define objects in JavaScript, reducing the need for repetitive code and improving readability.

Enhanced Object Literals are widely supported in modern JavaScript environments.

Computed Key Name

```
//Computed Property Names
const server = "server";

const serverInfo = {
  ["google" + server]: "server one",
  ["meta" + server]: "server two",
};
```

Function Shorthand

```
//Function Shorthand
const person = {
  fullName: "blame Gamer",
  job: "front-end",
  info() {
    console.log(`name is ${this.fullName}`);
    console.log(`job is ${this.job}`);
  },
};
```

Key Shorthand

```
//Key Shorthand
const serverName = "firebase";
const serverHost = "google";

//normal
const server = {
  serverName: serverName,
  serverHost: serverHost,
};

//using key shorthand
const server = {
  serverName,
  serverHost,
};
```



OOPs

Object Orient Programming System

Introduction Of OOPs

Object-oriented programming (OOP) is a programming paradigm that emphasizes the concept of objects, which can contain data and code to manipulate that data. JavaScript is a multi-paradigm language that supports OOP.

In JavaScript, objects can be created using either object literals or constructor functions. An object literal is a comma-separated list of name-value pairs wrapped in curly braces, while a constructor function is a special function that is used to create new objects.

In OOP, objects can have properties and methods. Properties are data members of an object, while methods are functions that operate on that data. In JavaScript, properties and methods can be added to objects using dot notation or bracket notation.

JavaScript also supports inheritance, which is a way for objects to inherit properties and methods from other objects. Inheritance can be achieved using either the prototype chain or class syntax.

The prototype chain is a way for objects to inherit properties and methods from their prototype object. Every object in JavaScript has a prototype object, which can be accessed using the `Object.getPrototypeOf()` method. Inheritance using the prototype chain is often called "prototypal inheritance".

The class syntax is a newer way of achieving inheritance in JavaScript, introduced in ES6. Classes are syntactical sugar over the existing prototype-based inheritance model, and provide a more familiar syntax for developers coming from other OOP languages.

Overall, JavaScript provides a powerful set of tools for implementing OOP concepts, including objects, properties, methods, inheritance, and more.

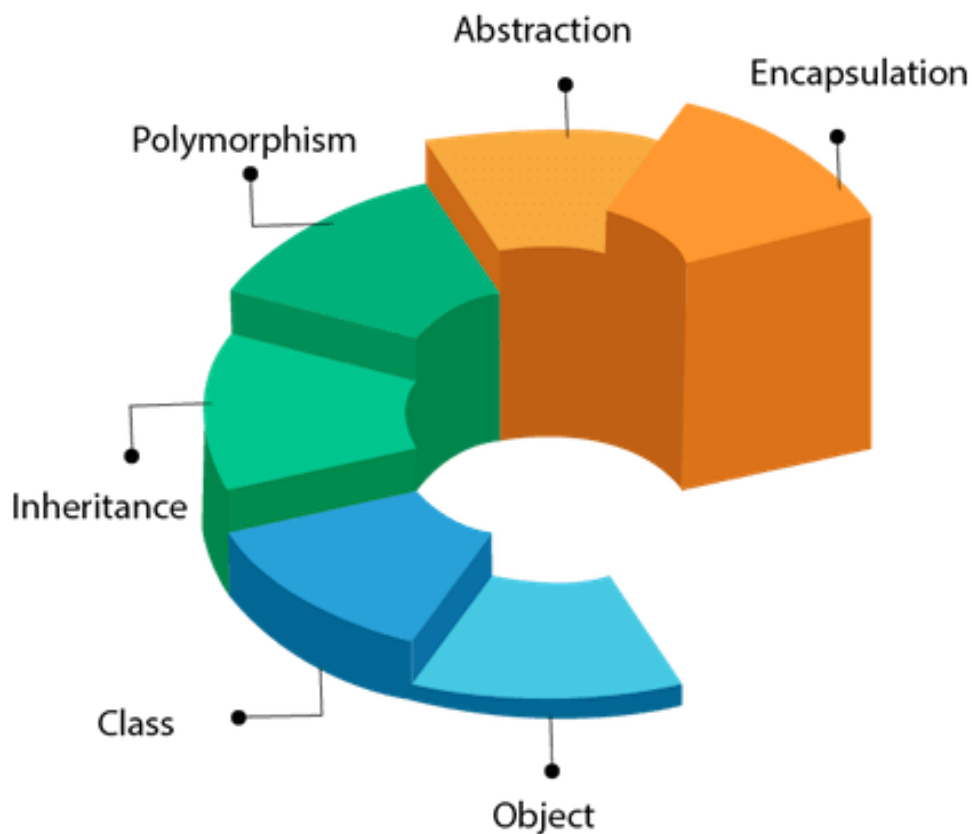
Four Principles Of OOPs

1.Abstraction

2.Encapsulation

3.Inheritance

4.Polymorphism



What is Class & Object?

In JavaScript, a **class** is a blueprint or template that defines the structure and behavior of objects. It describes the properties and methods that objects created from the class will have.

An **object** on the other hand, is an instance of a class. It is a concrete entity that is created based on the class definition. Objects have their own state (properties) and behavior (methods) defined by the class.

How To Create Class & Object

Three Ways To Create Class

1. Using Constructor Function
2. Using ES6 Classes
3. Using Object.Create

1.Create Class Using Constructor Function

In JavaScript, you can create classes and objects using constructor functions. A constructor function is a special function that is used to create and initialize objects of a class. To define a class using a constructor function, follow these steps:

1. Declare a function and give it a name that starts with a capital letter (by convention, class names are capitalized).
2. Inside the function, use the `this` keyword to refer to the current object being created.
3. Assign properties to the object using dot notation, where the property name is followed by a value.
4. Define methods by assigning functions to properties of the object.

```
//Create Class using constructor function
const PersonInfo = function (fn, ln, job) {

  this.firstName = fn;
  this.lastName = ln;
  this.profession = job;

  this.info = function () {
    console.log(`hello my name is ${this.fullName}`);
    console.log(`i am a ${this.lastName}`);
    console.log(`my profession is ${this.job}`);
  };
};

const blame = new PersonInfo("Blame", "Gamer", "front-end developer");
console.log(blame);
```

Constructor Function

In JavaScript, a **constructor function** is a regular function that is used to create and initialize objects. It serves as a blueprint or template for creating multiple objects with similar properties and behaviors.

New Operator

In JavaScript, the **new operator** is used to create an instance of an object or to invoke a constructor function. When you use the new operator with a constructor function, it performs the following steps:

1. Creates a new empty object.
2. Sets the prototype of the newly created object to the prototype of the constructor function.
3. Invokes the constructor function with the newly created object as the context (this).
4. If the constructor function does not explicitly return an object, it implicitly returns the newly created object.

Instances Of Class

In JavaScript, an **instance** refers to a specific object created from a class or constructor function. It is a concrete occurrence of the class, representing a unique entity with its own set of property values and methods. When you create an instance of a class or constructor function using the new keyword, you are essentially creating a new object that inherits the properties and methods defined in the class or constructor's prototype. Each instance has its own separate set of property values, allowing you to create multiple objects with similar behavior but different data.

Don't Use Arrow Function As Constructor Function

Arrow Function Problem

If you want to use an arrow function as a constructor, it's not possible in JavaScript. Arrow functions don't have their own this context, which is essential for constructor functions to properly initialize new objects.

Is it compulsory to name constructor function with capital letter?

In JavaScript, it is not compulsory to name a class or constructor function with a capital letter. However, it is a common convention and recommended best practice to use an initial capital letter for class and constructor function names. This helps distinguish them from regular functions and variables.

Use Object Properties

```
//blame is object
//so we can use object properties
console.log(blame.firstName);
console.log(blame.lastName);
console.log(blame.profession);
console.log(blame.info());

console.log(PersonInfo.prototype);
console.log(PersonInfo.__proto__.__proto__.__proto__);

console.log(blame instanceof PersonInfo);
console.log(blaze instanceof PersonInfo);
```

__proto__

__proto__ is a property in JavaScript that allows you to access the prototype of an object. It provides a way to access the internal prototype of an object and is not part of the official ECMAScript standard. Although widely supported in browsers, the use of **__proto__** is discouraged for several reasons, including potential performance impacts and potential compatibility issues.

2.Create Class Using ES6 Classes

In ES6, you can use the **class** keyword to declare a class. A class is a blueprint for creating objects with a predefined set of properties and methods. It provides a clean and concise syntax for object-oriented programming in JavaScript.

To create a class using ES6 classes, you follow these steps:

1. Declare the class using the **class** keyword, followed by the name of the class. For example, **class Person**.
2. Inside the class, you can define a constructor method using the **constructor** keyword. The constructor is a special method that is automatically called when you create a new instance of the class. It allows you to initialize the object's properties. You can pass parameters to the constructor to set the initial values of the object's properties.
3. Define other methods and properties of the class inside the class body. These methods and properties are shared among all instances of the class. You can directly define methods without using the **function** keyword.
4. To create an instance of the class, use the **new** keyword followed by the class name and parentheses. This calls the constructor method and creates a new object.

based on the class.

5. You can then access the properties and methods of the object using dot notation, like `objectName.property` or `objectName.method()`.

```
class PersonInfo {  
  
  constructor(fn, ln, job) {  
  
    this.FullName = fn;  
    this.lastName = ln;  
    this.profession = job;  
  
  }  
  
  fullInfo() {  
    console.log(`my name is ${this.FullName}`);  
    console.log(`i am a ${this.lastName}`);  
    console.log(`i am a ${this.profession} developer `);  
  }  
  
}  
  
const blame = new PersonInfo("blame", "gamer", "FE");  
  
//Now blame is object  
//So we can use object properties  
console.log(blame.FullName);  
console.log(blame.lastName);  
console.log(blame.fullInfo());  
console.log(blame instanceof PersonInfo);
```

3.Create Class Using Object.Create

```
const PersonInfo = {  
  jpeg() {  
    console.log("hello");  
  },  
  init(fn, ln) {  
    this.fullName = fn;  
    this.lastName = ln;  
  },  
};  
  
const blame = Object.create(PersonInfo);  
blame.init("blame", "gamer");  
console.log(blame);
```

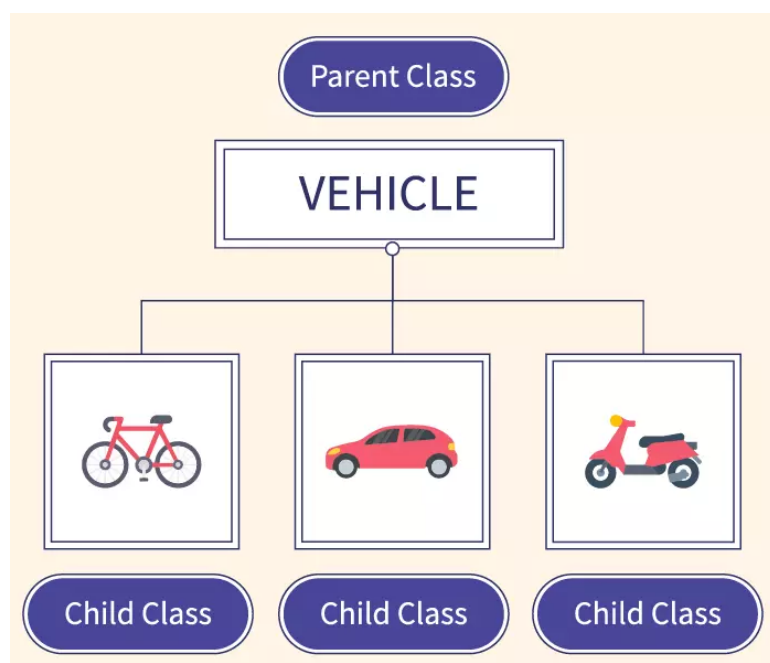
1. Inheritance or Class Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows objects to acquire the properties and behaviors of other objects. It is a mechanism that enables code reuse and establishes a hierarchical relationship between classes or objects.

Inheritance is based on the principle of creating new classes (derived classes) from existing classes (base or parent classes), where the derived classes inherit the attributes and methods of the parent class. The derived class can extend or override the inherited properties and methods, as well as define its own unique properties and methods.

General Example Of inheritance

Parent	Child 1	Child 2	Child 3
Height Hairs High IQ Powerful Memory skills Facial Expressions	height, skills, hairs etc	height, hairs, facial exp etc	High IQ, hairs, skills etc



Extend Keyword

In JavaScript, the extends keyword is used to establish inheritance between classes. It allows a derived class to inherit properties and methods from a parent (base) class. The extends keyword is part of the class syntax introduced in ECMAScript 2015 (ES6) and provides a more intuitive way to define and work with classes.

Super Method

The super() method refers to the parent class. By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

The super keyword in JavaScript is used to call the parent class constructor or parent class methods from within a subclass. It provides a way to access and invoke functionality from the parent class.

Classes provide "super" keyword for that.

1.super.method(...) to call a parent method.

2.super(...) to call a parent constructor (inside our constructor only).

Inheritance Example :

```
class Server {
  constructor() {
    this.serverName = "Firebase";
    this.serverLocation = "USA";
    this.serverMemory = "420 TB";
    this.request = 10;
  }
}

class CallServer extends Server {
  constructor(ln, dv) {
    super();
    this.yourLocation = ln;
    this.device = dv;
  }
}

const newRequest = new CallServer("india", "chrome laptop");
console.log(newRequest);
```

Another Inheritance Example :

An advance example to show how to achieve inheritance if parent class also have arguments.

For child class

Constructor(parent class parameters , child class parameters)

Super(parent class parameters)

```
class StudentPersonalInformation {
  constructor(fn, bn, yn, ln) {
    this.fullName = fn;
    this.branch = bn;
    this.year = yn;
    this.language = ln;
  }

  intro() {
    return `hello my name is ${this.fullName} and i am student
of ${this.year} year`;
  }
}

class Student extends StudentPersonalInformation {
  constructor(fn, bn, yn, ln, ad) {
    super(fn, bn, yn, ln);

    this.universityName = "AKTU";
    this.city = "bareilly";
    this.address = ad;
  }
}

const blame = new Student("blame pro", "it", 2023, "javascript",
"xyz road");
console.log(blame);
console.log(blame.intro());
```

Using super.method()

In JavaScript, `super.method()` is used to call the parent class's method from within a subclass. It allows you to invoke the overridden method in addition to executing the new functionality defined in the subclass. Here's an example to illustrate the usage of `super.method()`:

```
class ServerLogic {
  runServer() {
    console.log("running server logic");
    console.log("server is started");
  }
}

class CallingServer extends ServerLogic {
  start() {
    super.runServer();
  }
}

const request1 = new CallingServer();
console.log(request1.start());
```

class with only methods

if there is no properties in your class there is no need to use constructor function , simply don't write constructor function

Arrow functions have no super.method()

Arrow functions in JavaScript do not have their own super binding, and therefore they cannot access the `super` keyword to call the parent class's method. Arrow functions don't have their own `this` or `super`, so they transparently fit into the surrounding context

To access the parent class's method within an arrow function, you would need to store the parent's method reference in a variable before defining the arrow function and then use that variable within the arrow function.

Prototypal inheritance and prototype chain

In JavaScript, prototypal inheritance is a mechanism by which objects can inherit properties and methods from other objects. It is based on the concept of prototypes, where each object has an internal link to another object called its prototype. When accessing a property or method on an object, if it is not found on the object itself, JavaScript looks up the prototype chain to find it in the prototype object.

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

Person.prototype.greet = function () {
  console.log("Hello, my name is " + this.name);
};

const blame = new Person("blame");
console.log(blame);

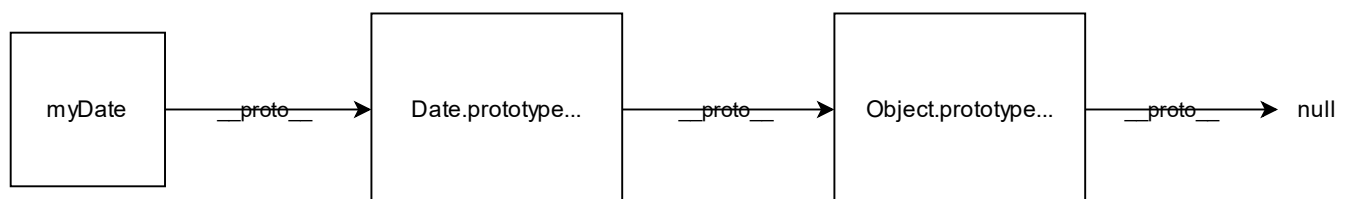
class Person {
  constructor(name) {
    this.name = name;
  }
}

Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};

const blame = new Person("blame");
console.log(blame);
```

Prototype Chain

In JavaScript, the null value is at the end of the prototype chain. When accessing properties or methods on an object, JavaScript continues to look up the prototype chain until it finds the property or method or until it reaches the end of the chain, which is null.



Viewer does not support full SVG 1.1

Inheritance chain

You can extend the inheritance chain as much as you want, setting parent, grandparent, great grandparent classes and so on.

```
class GreatGrandParent {
  constructor() {
    this.familyName = "GG Family Blame";
  }
}

class GrandParent extends GreatGrandParent {
  constructor() {
    super();
  }
}

class Parent extends GrandParent {
  constructor() {
    super();
  }
}

class Child extends Parent {
  constructor() {
    super();
  }
}

class GrandChild extends Child {
  constructor() {
    super();
  }
}

const baby1 = new GrandChild();
console.log(baby1);
```

check if baby1 instanceof GreatGrandParent is true or false

Overriding a method

In JavaScript, you can override a method by redefining it in the child class or object. Here's an example that demonstrates method overriding:

```
class Car {
  constructor(carName, carNumber) {
    this.carName = carName;
    this.carNumber = carNumber;
  }

  superSpeed() {
    console.log("normal car");
    console.log("no super speed");
  }
}

class SuperCar extends Car {
  constructor(carName, carNumber, superCarName, superCarModel) {
    super(carName, carNumber);
    this.superCarName = superCarName;
    this.superCarModel = superCarModel;
  }

  superSpeed() {
    console.log("super car");
    console.log("super car speed");
  }
}

const car1 = new SuperCar("normal car", 2214, "bmw", "super fast car");
console.log(car1.superSpeed());
```

Static method

In JavaScript, a static method is a method that belongs to a class rather than an instance of that class. Static methods are called directly on the class itself, rather than on an object created from the class. They are useful for defining utility functions or operations that are not dependent on any specific instance of the class.

Static class methods are defined on the class itself. You cannot call a static method on an object, only on an object class.

```
class ServerLogic {
  constructor() {
    this.serverName = "Firebase";
    this.serverLocation = "USA";
  }

  static runServer() {
    console.log("running server logic");
    console.log("server is started");
  }

  //static property
  static serverLink = "www.firebase.com";
}

const request1 = new ServerLogic();
console.log(request1);
console.log(request1.runServer()); //error (static method can't
be used on object)
console.log(ServerLogic.runServer());
```

Static properties

Static properties are new and not use widely. Static properties are not yet supported in JavaScript. However, there are discussions and proposals for adding static properties to the language in the future.

Getter and Setter

In JavaScript, getter and setter methods are used to define object properties with custom behavior. They allow you to control how a property is accessed and modified. Here's how you can use getter and setter methods in JavaScript:

Getter Method: A getter method is used to retrieve the value of a property. It is defined using the `get` keyword followed by the method name and an empty set of parentheses.

Setter Method: A setter method is used to set the value of a property. It is defined using the `set` keyword followed by the method name and a single parameter.

```
class FlyingOfficer {
  constructor(fn) {
    this.fullName = fn;
  }

  get getName() {
    return this.fullName.toUpperCase();
  }

  set setName(n) {
    this.fullName = n.toUpperCase();
  }
}

//getter is used as property
//so don't use () for getter
const officer1 = new FlyingOfficer("blame");
console.log(officer1.getName);

//setter
officer1.setName = "blade";
console.log(officer1);
```

2.Encapsulation

Encapsulation is a fundamental principle of object-oriented programming that involves bundling data and the methods that operate on that data into a single unit, known as an object. It aims to hide the internal state and implementation details of an object, providing controlled access to the object's properties and behaviors through a well-defined interface.

Encapsulation is another key concept in OOP, and it stands for an object's capacity to "decide" which information it exposes to "the outside" and which it doesn't.

Encapsulation is implemented through public and private properties and methods. In JavaScript, all objects' properties and methods are public by default. "Public" just means we can access an object's property/method from outside its own body:

We first need to declare the private property, always using the '#' symbol as the start of its name.

```
class Server {
  #serverApiKey;

  constructor() {
    this.serverName = "firebase";
    this.#serverApiKey = 46525145;
  }

  start() {
    console.log(`server is starting with api key as
    ${this.#serverApiKey}`);
  }

  serverLogic() {
    console.log("logic of server");
  }
}

const request1 = new Server();
console.log(request1.serverApiKey); //undefined
console.log(request1.start());
```

3. Abstraction

Abstraction in JavaScript refers to the concept of hiding implementation details and exposing only essential features or functionalities of an object or module. It allows you to work with higher-level concepts and interact with objects using a simplified interface, without needing to know the intricate details of how they are implemented.

Abstraction is a principle that says that a class should only represent information that is relevant to the problem's context. In plain English, only expose to the outside the properties and methods that you're going to use. If it's not needed, don't expose it. This principle is closely related to encapsulation, as we can use public and private properties/methods to decide what gets exposed and what doesn't.

For example: phone

phone

battery percentage

screenBrightness

volume

wallpaper

batteryLogic()

temperature()

voltage()

```
class ServerLogic {
  serverLogic() {
    console.log("logic of server");
    console.log("server is started");
  }
}

class Server extends ServerLogic {
  constructor() {
    super();
    this.serverName = "firebase";
    this.serverApiKey = 46525145;
  }

  start() {
    console.log(`server is starting`);
  }
}

const request1 = new Server();
console.log(request1.serverLogic());
```

4. Polymorphism

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different types to be treated as instances of a common superclass or interface. It enables code to be written in a way that is more generic and flexible, as it can operate on objects with different behaviors without needing to know their specific types.

Polymorphism means "many forms" and is actually a simple concept. It's the ability of one method to return different values according to certain conditions.

In JavaScript, polymorphism can be achieved through inheritance and method overriding.

```

class Animal {
  constructor(animalname) {
    this.animalName = animalname;
  }

  animalSound() {
    console.log(`${this.animalName} is making sound`);
  }
}

class Dog extends Animal {
  constructor(animalname) {
    super(animalname);
  }

  animalSound() {
    console.log(`${this.animalName} is making sound woof woof`);
  }
}

class Cat extends Animal {
  constructor(animalname) {
    super(animalname);
  }

  animalSound() {
    console.log(`${this.animalName} is making sound meow meow`);
  }
}

const dog1 = new Dog("doggie");
const cat1 = new Cat("catie");

console.log(dog1.animalSound());
console.log(cat1.animalSound());

```

End Of Object Orient Programming

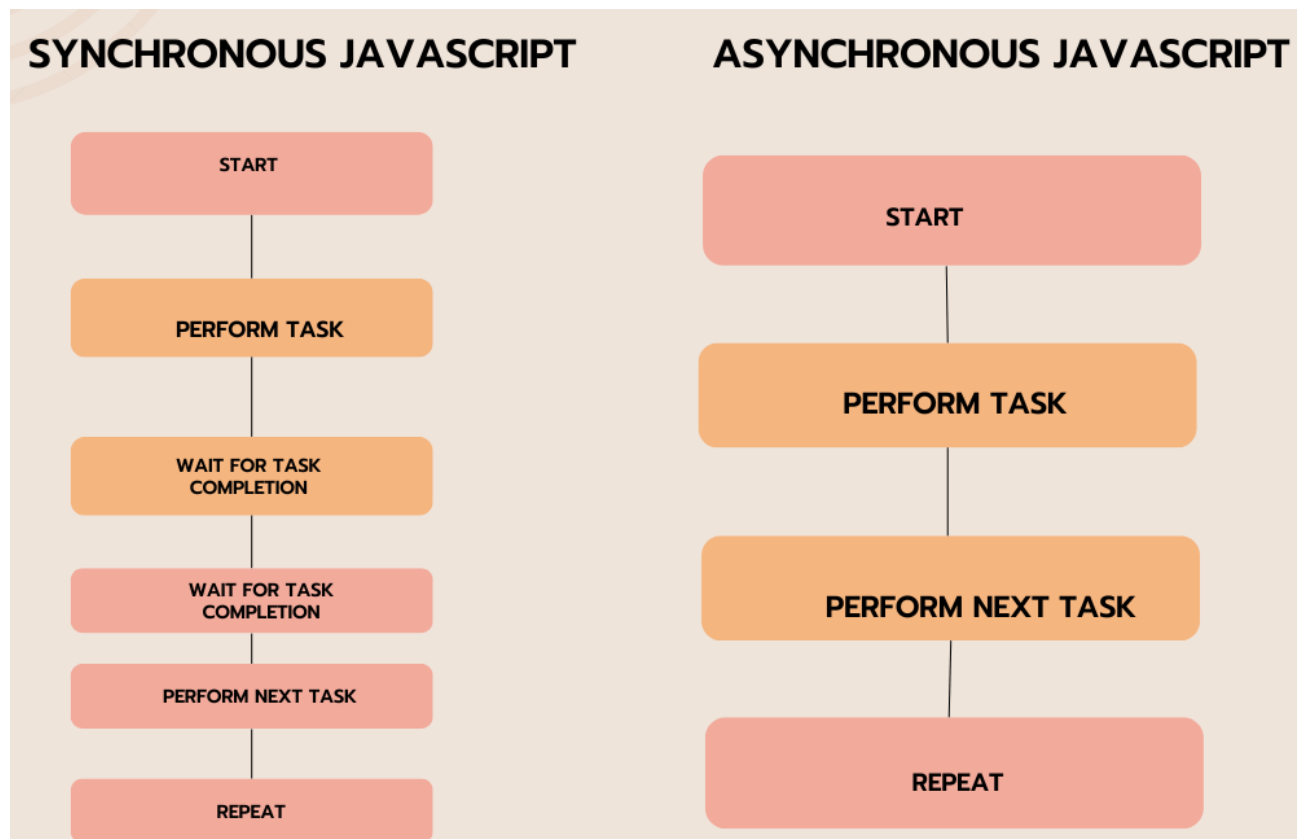
Asynchronous Javascript Start From Here



**True Power Of Javascript
True Power Of Developers**

Synchronous and Asynchronous Code

In JavaScript, synchronous and asynchronous code execution can be achieved using different techniques and language features. Let's explore both synchronous and asynchronous code examples:



Synchronous Code :

Synchronous code executes sequentially, blocking the execution until a task is completed before moving on to the next one.

Start Synchronous

Task 1 Synchronous

Task 2 Synchronous

Task 3

End

```
//Synchronous : Code is run line by line
const string = "Synchronous code : ";
const string2 = "code is run";
```

```
const string3 = "line by line";  
console.log(string);  
console.log(string2);  
console.log(string3);  
console.log("example end here");
```

Asynchronous Code :

Asynchronous code allows tasks to be executed concurrently or non-blockingly. JavaScript provides several mechanisms for writing asynchronous code, such as callbacks, promises, and async/await.:

Start

End

Asynchronous Task

Callback executed

```
//Asynchronous : code is not run line by line  
const string = "Asynchronous code : ";  
const string2 = "code is not run";  
const string3 = "line by line";  
setTimeout(() => console.log(string, string2, string3), 3000);  
console.log("example start here");
```

is javascript synchronous or asynchronous?

JavaScript itself is not strictly synchronous or asynchronous—it supports both synchronous and asynchronous code execution.

In JavaScript, by default, code execution is synchronous, meaning that statements are executed in a sequential and blocking manner. Each statement is executed one after the other, and the program waits for each statement to complete before moving on to the next.

However, JavaScript also provides mechanisms for handling asynchronous operations. These mechanisms, such as callbacks, promises, and async/await, allow you to write asynchronous code and manage tasks that take time to complete without blocking the execution flow. Asynchronous operations, like making API

calls or reading files, can be initiated, and the program can continue executing other code while waiting for the results of those operations.

So, JavaScript can be both synchronous and asynchronous depending on how you write your code and which mechanisms you use. It supports synchronous execution by default and provides ways to handle asynchronous tasks effectively.

Event Loop

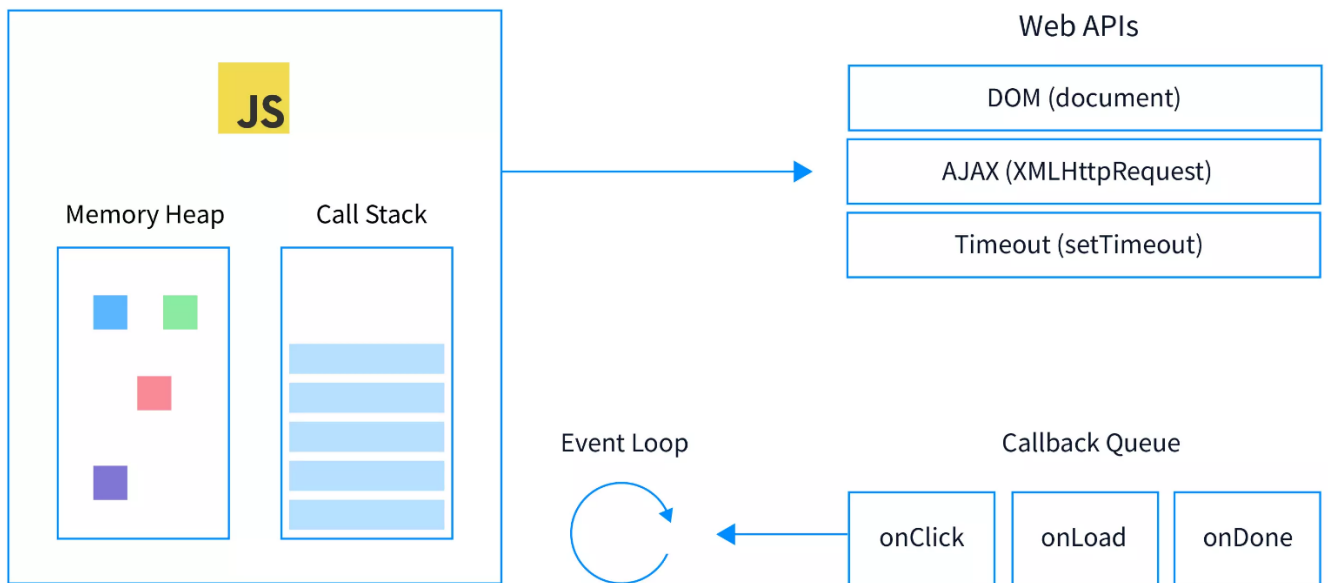
The event loop is a key part of how JavaScript handles asynchronous code. In simple terms, it's like a "traffic controller" that manages the flow of code execution in JavaScript.

When a script is run in a web browser or a Node.js environment, JavaScript code is executed on a single thread. This means that only one set of instructions can be executed at a time. If a function takes a long time to execute, it will block the thread and prevent any other code from running until it's finished.

To avoid this problem, JavaScript uses an event loop. When you run an asynchronous function in JavaScript (e.g., using a `setTimeout` or a `fetch` call), it's placed in a queue and the event loop waits for it to complete. In the meantime, the event loop continues to execute other code that's waiting to be executed.

Once an asynchronous function is complete, it's placed back in the event queue and the event loop checks the queue to see if there are any other functions waiting to be executed. If there are, it dequeues them and starts executing them one by one. This allows JavaScript to handle multiple requests and operations at the same time without blocking the thread, which can make web applications feel faster and more responsive.

In summary, the event loop is a mechanism that allows JavaScript to handle asynchronous code in a non-blocking way by placing functions in a queue and executing them one by one when they're ready.



AJAX

AJAX (Asynchronous JavaScript and XML) is a technique used to send and retrieve data from a server without reloading the entire web page. It enables the exchange of data between a web browser and a server asynchronously, in the background.

The term "XML" is often used in the name AJAX because XML was commonly used as the data format for the exchanged data. However, nowadays, JSON (JavaScript Object Notation) is more commonly used instead of XML due to its lighter weight and easier integration with JavaScript.

AJAX leverages the XMLHttpRequest (XHR) object in JavaScript to make asynchronous HTTP requests to a server. These requests can be used to fetch data, send data, or update parts of a web page dynamically.

API (Application Programming Interface)

An API (Application Programming Interface) is a set of rules and protocols that allow different software applications to communicate and interact with each other. It defines the methods, data formats, and rules that developers can use to request and exchange information between different systems. APIs act as intermediaries, providing a way for applications to access certain functionalities or data of other software systems, such as web services, libraries, operating systems, or databases. They enable developers to build upon existing systems, integrate different services,

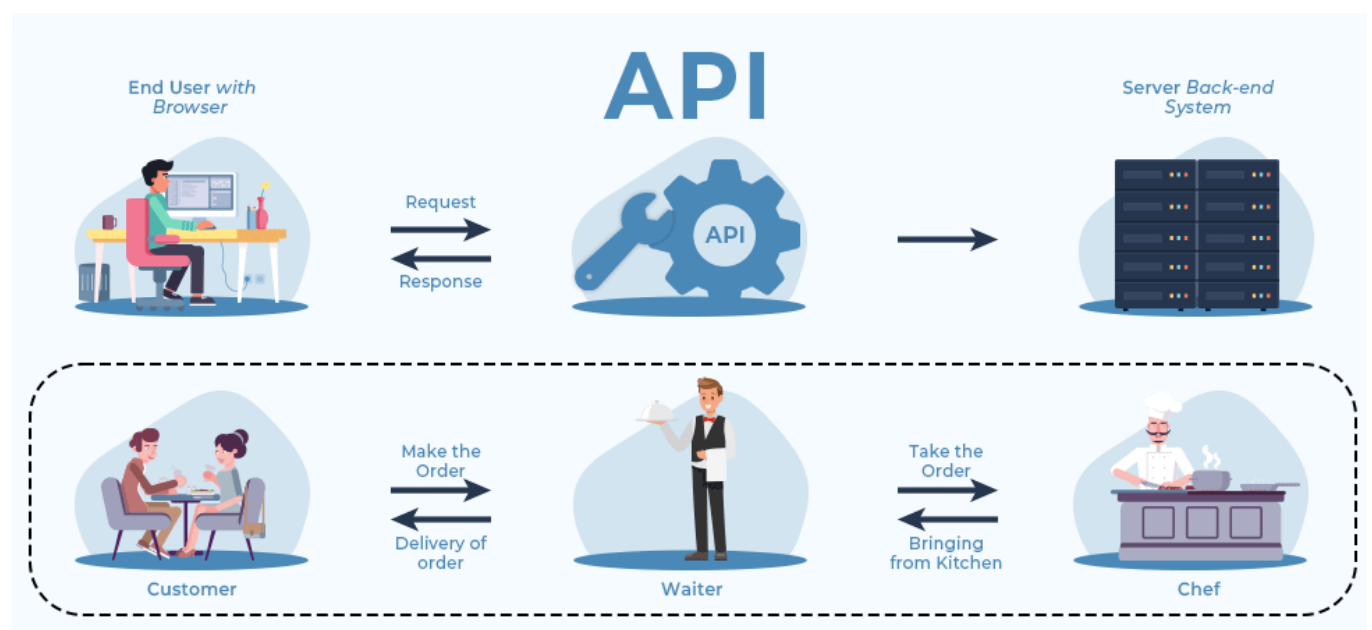
and create new applications by leveraging the functionalities exposed through the API.

An API is like a set of rules that allows computer programs to talk to each other, share information, and perform specific actions. It enables different applications to work together and provide you with useful services and features.

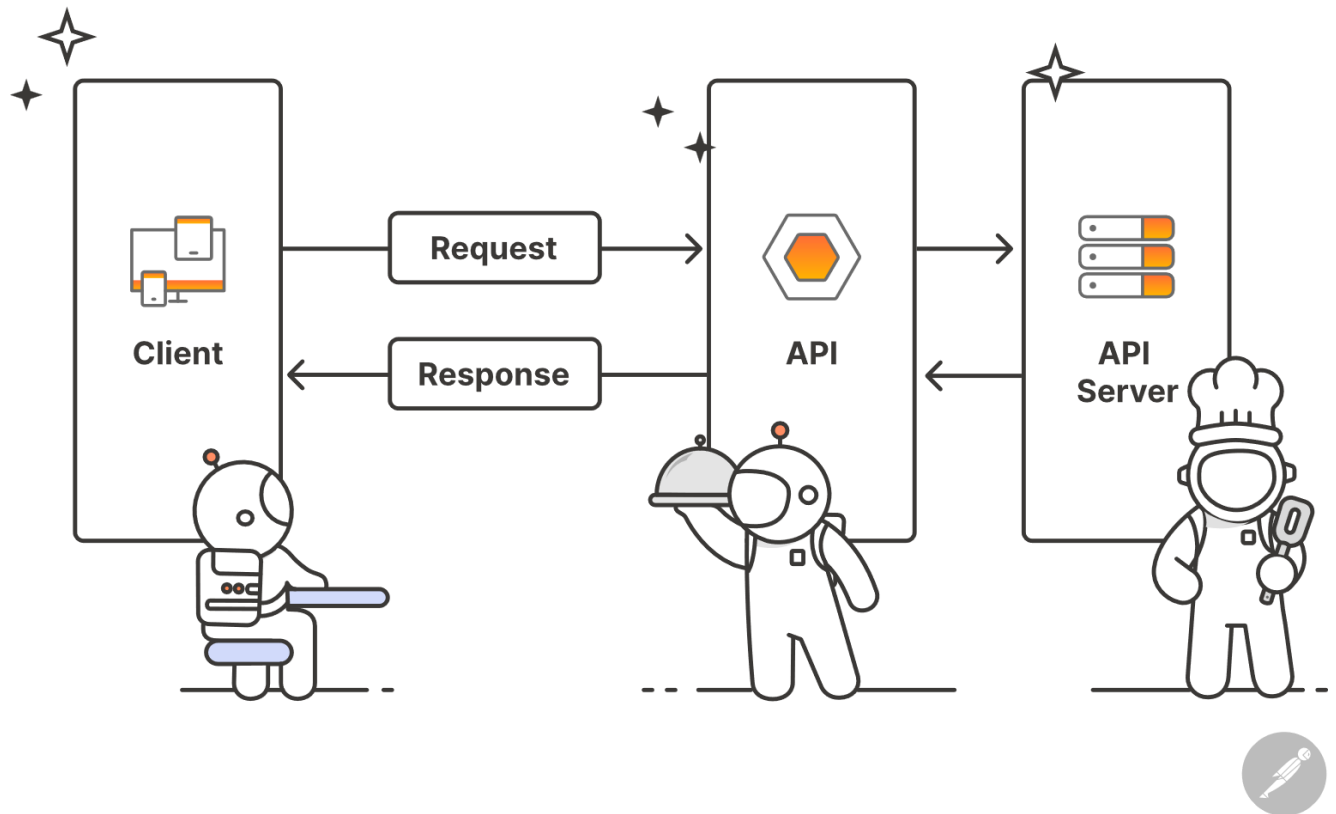
Example of api : airline ticket booking



api : An API provides a way for two computers or software programs to communicate and interact with each other.



Best Example to learn api



Working with api

1.XML

2.Fetch api

Install thunderbolt or postman extension in vscode for testing api

Free API Link : <https://github.com/public-apis/public-apis>

Taking example of rest countries api from above link

<https://restcountries.com/v3.1/name/{name}>

Different method in api

methods are part of the HTTP protocol and are used to perform different types of operations on resources. The choice of method depends on the desired operation and the semantics of the API or server you are working with. It's important to adhere to the proper use of these methods to ensure correct and consistent behavior when interacting with web services and APIs.

GET : The GET method is used to retrieve data from a server. When a GET request is made, the server sends back the requested data in the response body. It is a safe and idempotent method, meaning multiple identical GET requests will have the same effect and do not modify the server's state.

POST : The POST method is used to send data to the server, typically to create a new resource. The data is included in the request body. POST requests can trigger side effects on the server, such as storing data in a database or initiating a transaction. Unlike GET, POST requests are not idempotent, as multiple requests may result in multiple resource creations.

PUT : The PUT method is used to update or replace an existing resource on the server. It requires the complete representation of the resource to be sent in the request body. If the resource doesn't exist, PUT may create it. PUT requests are idempotent, meaning multiple identical requests will have the same effect.

DELETE : The DELETE method is used to delete a resource identified by a specific URL. It performs the deletion of the resource on the server. DELETE requests are idempotent, as multiple identical requests will have the same effect.

PATCH : The PATCH method is used to partially update an existing resource. It requires sending only the specific changes to be applied, rather than sending the complete representation of the resource. PATCH requests are not always idempotent, as multiple identical requests may produce different results if applied multiple times.

1.XHR or XMLHttpRequest Method

An XML request, also known as an XMLHttpRequest or XHR request, is a mechanism in web development that allows client-side JavaScript code to asynchronously communicate with a server. It enables the exchange of data between the client and the server without requiring a full page reload.

XML requests are commonly used to retrieve data from a server, submit form data, or interact with web APIs. The client-side code initiates an XML request by creating an XMLHttpRequest object, specifying the HTTP method (such as GET or POST)

and the URL of the server-side resource or API endpoint. The request can include headers, request parameters, and a request body containing data in XML format.

Once the request is sent, the server processes it and generates a response. The response may contain various types of data, such as XML, JSON, HTML, or plain text. The client-side code can handle the response by defining event handlers for events like ``onload``, ``onerror``, or ``onprogress``. The response data can then be used to update the user interface, perform further operations, or make subsequent requests.

XML requests are typically performed asynchronously, allowing other JavaScript code to continue executing while waiting for the response. This asynchronous behavior improves the user experience by preventing the browser from becoming unresponsive during long-running operations. However, synchronous XML requests can also be made when necessary.

Overall, XML requests provide a means for client-side JavaScript to interact with servers, exchange data, and update web pages dynamically, without requiring a full page reload.

In the context of XML requests made with XMLHttpRequest (XHR) in JavaScript, there are several events that you can handle to track the progress and status of the request. Here are some commonly used events:

1. `onreadystatechange`: This event is triggered whenever the ``readyState`` property of the XHR object changes. You can define a handler to perform actions based on the current state of the request.
2. `onload`: This event is fired when the request is successfully completed and the response has been received. You can process the response data inside this event handler.
3. `onerror`: This event is triggered if an error occurs during the request, such as a network failure or a server error. You can handle the error condition and take appropriate action.
4. `onprogress`: This event is periodically triggered during the progress of the request, providing information about the amount of data transferred. It allows you to track the progress and update the user interface accordingly.
5. `ontimeout`: This event is fired if the request exceeds the specified timeout duration. You can handle this event to perform actions when the request times out.

Country api example


```

//Country api example
//api url : https://restcountries.com/v3.1/name/{name}

//"https://restcountries.com/v3.1/name/russia" => this api
endpoint will give you russia information

//Create a new XMLHttpRequest object
let request = new XMLHttpRequest();

//Configure it
request.open("GET",
"https://restcountries.com/v3.1/name/russia");
// request.responseType = "json";

request.onload = function () {
    console.log(request.response);
    console.log(JSON.parse(request.response));
    console.log(request.status);

    if (request.status >= 400) {
        console.log("failed");
    } else {
        console.log(request.response);
    }
};

request.onerror = function () {
    console.log("error");
};

//Send the request over the network
request.send();

```

JSON fake api example

```

//fake json api example
//api url : https://jsonplaceholder.typicode.com/users

//Create a new XMLHttpRequest object
let request = new XMLHttpRequest();

//Configure it

```

```

request.open("POST",
"https://jsonplaceholder.typicode.com/users");
// request.responseType = "json";

request.setRequestHeader("Content-Type", "application/json");

request.onload = function () {
    console.log(request.response);
    console.log(JSON.parse(request.response));
    console.log(request.status);

    if (request.status >= 400) {
        console.log("failed");
    } else {
        console.log(request.response);
    }
};

request.onerror = function () {
    console.log("error");
};

const body = {
    name: "testing data",
    power: "78 hrs",
};

//Send the request over the network
request.send(JSON.stringify(body));

```

Header in api

In JavaScript, when making HTTP requests to an API, the "header" refers to the additional information sent along with the request. Headers are used to provide metadata about the request or to customize the behavior of the server handling the request. Headers consist of key-value pairs, where the key represents the specific header field and the value contains the associated data. Common headers used in API requests include:

Content-Type: Specifies the type of data being sent in the request body, such as "application/json" for JSON data or "application/x-www-form-urlencoded" for form data.

Authorization: Provides authentication credentials for the request, often using

tokens or API keys.

Accept: Informs the server about the preferred response format, such as "application/json" or "text/html".

User-Agent: Identifies the client making the request, usually including details about the browser or application.

Cache-Control: Controls caching behavior, indicating whether the response should be cached or not.

Cookie: Contains any cookies associated with the request.

2.Fetch Api

The Fetch API is a modern interface that allows you to make HTTP requests to servers from web browsers. If you have worked with XMLHttpRequest (XHR) object, the Fetch API can perform all the tasks as the XHR object does. In addition, the Fetch API is much simpler and cleaner. It uses the Promise to deliver more flexible features to make requests to servers from the web browsers.

```
//fetch syntax
fetch("api-url")
  .then((response) => response.text())
  .then((data) => console.log(data));
```

Promises

async/ await

Currying
Debouncing
Polyfill