Introduction to JAVASCRIPT

Need for JavaScript

Limitations and Constraints of the Webpage Created with HTML & CSS:

- This simple static HTML currently lacks responsiveness but includes images, text, buttons, and other UI features.
- The website needs to be utilized from a business or a customer standpoint.

JavaScript is useful because it provides the following advantages:

- Our system is lightning-fast because it runs code locally instead of contacting the server.
- Create highly responsive interfaces and elevate the user experience with ease.
- JavaScript does not require a compilation step. Instead, an interpreter in the browser reads over the JavaScript code, interprets each line, and runs it.
- Enhance functionality on the fly without server latency, providing a seamless experience for your users.

What makes them so well-loved by programmers?

- As of today, there are 1.9 billion websites on the Internet, with 95% of them utilizing JavaScript in some capacity.
- To improve the functionality of certain websites and web applications, try disabling JavaScript access on your web browser. JavaScript is responsible for managing the interactivity of the web, and turning it off may cause some sites to appear less engaging.
- Additionally, developers appreciate it for its user-friendly nature, making it easy to learn and become proficient in. Even with no prior coding experience, one can create stunning projects with the knowledge of JavaScript. For businesses, this translates to a reliable pool of competent JavaScript developers always at the ready.

Other than simplicity, JavaScript is also known for its speed and versatility. It can be used for the following programming projects:

- Add interactive elements to your website
- Develop web and mobile applications
- Create web servers
- Develop games

Why is JavaScript Popular?

1. **Versatility**: JavaScript is primarily used for front-end development, but it can also be used for server-side programming with Node.js. This flexibility allows developers to work across the full stack with a single language.
2. **Browser compatibility:** JavaScript is supported by all modern web browsers, making it an essential tool for creating interactive and dynamic websites that work across different platforms and devices.
3. **Ease of learning:** JavaScript has a relatively simple syntax and is considered easier to learn compared to some other programming languages.
4. **Large community and ecosystem:** JavaScript has a massive community of developers, which means there are abundant resources, tutorials, and tools available for learning and problem-solving. Additionally, there are numerous libraries and frameworks, such as React, Angular, and Vue.js, that make development faster and more efficient.
5. **Regular updates and improvements:** JavaScript is continuously evolving, with new features being added to the language regularly through the ECMAScript standard. This ensures that JavaScript stays up-to-date with modern programming practices and techniques.
6. **Asynchronous programming:** JavaScript supports asynchronous programming, which is essential for handling multiple tasks simultaneously, such as handling user input or fetching data from a server without blocking the main thread.

JavaScript Engine

Each web browser has its own JavaScript engine that supports JavaScript scripts so that they can function properly. The main task of a JavaScript engine is to take JavaScript code and convert it into optimized code, fast that can be interpreted by a browser. Here are the names of the JavaScript engines used in some of the most popular web browsers:

- Chrome: V8
- Firefox: SpiderMonkey
- Safari: JavaScript Core
- Microsoft Edge/Internet Explorer: Chakra/ChakraCore.

Try the following code on JSFIDDLE

Prompt

```
const name = prompt("What's your name");
console.log(name);
```

Confirm

```
const result = confirm("Are you sure ?");
console.log(result);
```

Alert

```
alert("Hello World");
```

prompt sync

readline

```
const readline = require('readline').createInterface({
  input: process.stdin,
  output: process.stdout
});

readline.question('Enter two numbers separated by space: ', (input) ⇒
{
  const [num1, num2] = input.split(' ').map(Number);
  console.log(num1 + num2);
  readline.close();
});
```

What is a variable ?

It is a container to hold a value

var, let and const

Declaration (Multiple same declaration not possible)

```
var a = 100;
var a = 100;
var a = 101;

a = 200;

console.log(a);
```

Declaration (Multiple same declaration not possible)

```
let a = 100;
let a = 100;
let a = 101;

console.log(a);  // Error
```

const

```
const a;
a = 100;

console.log(a)  // error
```

var is function scoped

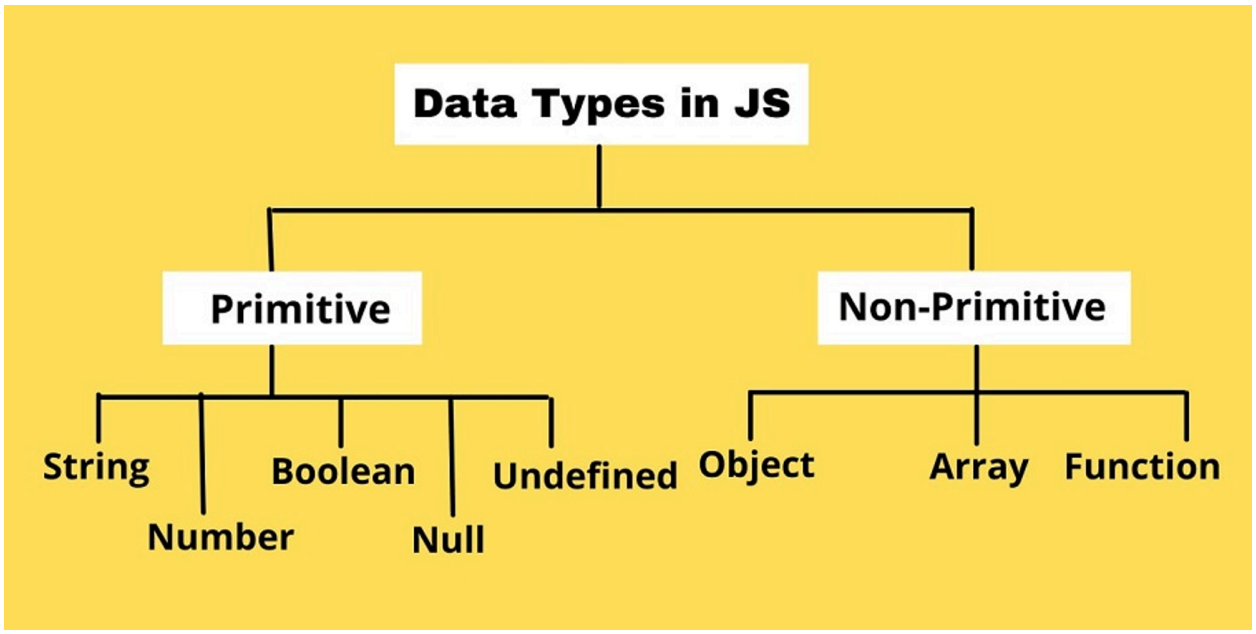
```
var name = "Ramesh";

function test(){
    var name = "Vicky"
    console.log("Inside house name : ",name);
}

test()
console.log("Inside house name : ",name);
```

let and const are block scoped

```
//let or const
let name = "Ramesh";

if(true){
    let name = "Vicky"
    console.log("Inside house name : ",name);
}

console.log("Inside house name : ",name);
```

How to display using string literal (${})

Javascript Data Types



BigInt :

JavaScript integers are only accurate up to 15 digits:

In JavaScript, all numbers are stored in a 64-bit floating-point format (IEEE 754 standard).

With this standard, large integer cannot be exactly represented and will be rounded.

Because of this, JavaScript can only safely represent integers:

Up to 9007199254740991 +(253-1)

and

Down to -9007199254740991 -(253-1).

Integer values outside this range lose precision.

To create a BigInt, append n to the end of an integer or call BigInt():

```
let x = 1234567890123456789012345n;
let y = BigInt(1234567890123456789012345)

console.log(typeof x)  // BigInt
console.log(typeof y)  // BigInt
```

Symbol

Symbol is a new primitive data type

The JavaScript ES6 introduced a new primitive data type called `Symbol`.

Symbols are immutable (cannot be changed) and are unique. For example,

```
// How to create a symbol

   const x = Symbol()
   typeof x; // symbol
```

Symbols are unique

```
   const value1 = Symbol('hello');
   const value2 = Symbol('hello');
   console.log(value1 === value2); // false

   var a1 = 100
   var a2 = 100

   console.log(a1 === a2) // false

const x = Symbol('hey');
const y = Symbol('hey');

console.log(x === y) // false
```

To access the description of a symbol, we use the `.` operator. For example,

```
const x = Symbol('hey');
console.log(x.description); // hey
```

Symbols are unique and immutable

```
var x = Symbol('hey');
x = Symbol("hello");

console.log(x.description)
```

The `for...in` loop does not iterate over Symbolic properties. For example,

```
let id = Symbol("id");

let person = {
    name: "Jack",
    age: 25,
    [id]: 12
};

// using for...in
for (let key in person) {
    console.log(key);
}
```

Benefit of Using Symbols in Object

If the same code snippet is used in various programs, then it is better to use `Symbols` in the object key. It's because you can use the same key name in different codes and avoid duplication issues. For example,

```
let person = {
    name: "Jack"
};
// creating Symbol
let id = Symbol("id");

// adding symbol as a key
person[id] = 12;
```

In the above program, if the `person` object is also used by another program, then you wouldn't want to add a property that can be accessed or changed by another program. Hence

by using `Symbol`, you create a unique property that you can use.

Now, if the other program also needs to use a property named id, just add a Symbol named `id` and there won't be duplication issues. For example,

index.js

```
const jsObj = {
  name : "jack"
}
module.exports = {jsObj}
```

main.js

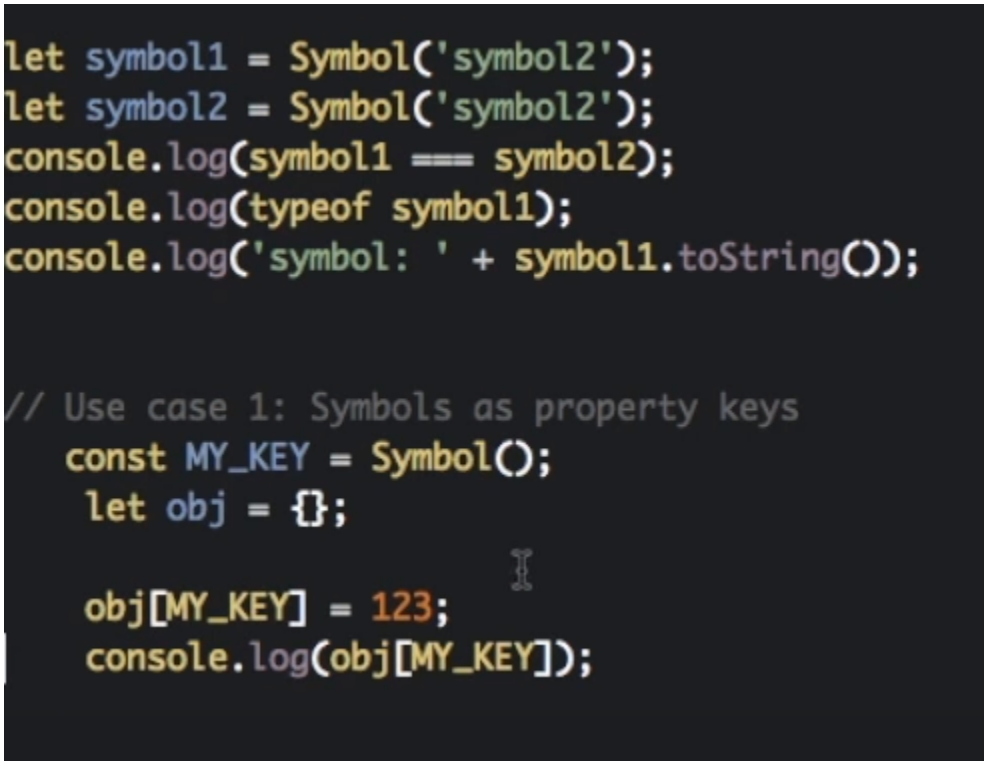
```
const obj = require('./index')
let id = Symbol("id");
obj[id] = 12
console.log(obj[id])
```

```
const obj2 = require('./index')
let id = Symbol("id");
```

```
obj2[id] = "Another Value"

console.log(obj2[id])
```

Using string will overwrite

```
let person = {

    name: "Jack"

};



// using string as key

person.id = 12;

console.log(person.id); // 12



// Another program overwrites value

person.id = 'Another value';

console.log(person.id); // Another value
```


let symbol1 = Symbol('symbol2');
let symbol2 = Symbol('symbol2');
console.log(symbol1 === symbol2);
console.log(typeof symbol1);
console.log('symbol: ' + symbol1.toString());
// Use case 1: Symbols as property keys
const MY_KEY = Symbol();
let obj = {};
obj[MY_KEY] = 123;
console.log(obj[MY_KEY]);

```
// Use case 2: constants representing concepts
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');

function getComplement(color) {
    switch (color) {
        case COLOR_RED:
            return COLOR_GREEN;
        case COLOR_ORANGE:
            return COLOR_BLUE;
        case COLOR_YELLOW:
            return COLOR_VIOLET;
        case COLOR_GREEN:
            return COLOR_RED;
        case COLOR_BLUE:
            return COLOR_ORANGE;
        case COLOR_VIOLET:
            return COLOR_YELLOW;
        default:
            throw new Exception('Unknown color: '+color);
    }
}
```

1. **Array**: An Array is a collection of values or elements, which can be of any data type that is stored in a contiguous memory location and accessed using an index.

```
var names = [ “Mayank”, “Shubham”, “Amrita”]; // Array of
stringsvar ages = [ 30, 29, 33 ] ; // Array of numbers
```

2. **Objects**: An object refers to a set of properties consisting of a key and a corresponding value. These values can include primitive data types, functions, or even other objects.

```
var student = {
  roll_no: 34,
  name: "Mayank",
  age: 27,
  city: "Delhi"
};
```

3. **Functions**: A function is a block of code used to perform a specific task or action and can be reused in different parts of a program.

```
function sum(a, b) {
  return a + b
}

console.log(sum(3, 4)); // 7
console.log(sum(5, 6)); //11
```

- ➢ Interpreter vs compiler
- ➢ Dynamically typed vs Statically typed

Statically typed vs Dynamically Typed languages

Implicit vs explicit type conversion

Difference between null and undefined

```
var length=19;
var breadth;

var area=length*breadth;
console.log(area);//NaN

var length2=19;
var breadth2=null;

var area2=length2*breadth2;
console.log(area2);//0
```

Null vs Undefined

```
var z = null;
var y;  // only declared not assigned any value

console.log(typeof z)  //object

console.log(y)  //undefined
console.log(typeof y) //undefined

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

In JavaScript, the == operator performs type coercion, meaning it tries to convert operands to the same type before comparing them. On the other hand, the === operator, known as the strict equality operator, does not perform type coercion and checks both value and type equality.

Let's break down your examples:

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

Here, null and undefined are considered loosely equal because both represent absence of value in JavaScript. When using the == operator, JavaScript performs type coercion, converting both null and undefined to a common type (in this case, both are treated as equal when converted to null), so the comparison returns true.

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

In this case, === checks strict equality, meaning it compares both the value and the type. Since null and undefined are of different types (null is an object and undefined is of type undefined), the comparison returns false.

Here are examples to illustrate:

javascript

```
console.log(typeof null); // "object"
console.log(typeof undefined); // "undefined"

console.log(null == undefined); // true, because both are treated as null

console.log(null === undefined); // false, because they are of different types
```

This behavior is important to understand when writing JavaScript code, as it can lead to unexpected results if not handled properly, especially when dealing with comparisons.

If any value is not assigned then you can assign null to that variable

A NULL value represents the absence of a value for a record in a field. An empty value is a value with no significant data in it.

a NULL means that there is no value, we're looking at a **blank/empty cell**, and 0 means the value itself is 0. Considering there is a difference between NULL and 0

```
var x = null;
var z = 5/x;
console.log(z)   //infinity
```

```
var a = null;
console.log(Number(a));
```

0/empty/NaN/Undefined/Null

Boolean of all the above value is false

What is NaN ?

By definition, NaN is the return value from operations which have an undefined numerical result. Hence why, in JavaScript, aside from being part of the global object, it is also part of the Number object: Number. NaN. It is still a numeric data type , but it is undefined as a real number .

In JavaScript, **NaN** is short for "**Not-a-Number**". In JavaScript, **NaN** is a number that is not a legal number.

```
var z = NaN;
console.log(typeof z)

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

https://dev.to/amrtaher1234/why-nan-nan-returns-false-in-javascript-14o6

```
Why NaN == NaN or NaN === NaN is False ?
let a = "5"
let b = 5

let x =  Number("Hello")
let y = 6/"Hello"

console.log(x)
console.log(y)

console.log(a == b)  // true
```

```
console.log(a ≡ b) // false

console.log(x = y) // false
console.log(x ≡ y) // false
```

The scope of var, let, and const can greatly affect how your code behaves in repeated functions. Understanding these differences is crucial to writing efficient and reliable code.

Below is an example of a breakdown of his method:

1. var:

Function scope: accessible from the entire function, even in recursive calls.

Example: This code will not calculate factorials since each callback has a different factorial.

JavaScript Code

```
function FactorialVar(n) {
if (n ≡ 0) {
return 1;
} else {
return n * FactorialVar(n); // Same var used
}
console.log(factorialVar(5)); // Output is 20 instead of 120
```

Please use your code carefully.

2. let:

Block scope: Accessible only from the block in which it is declared (such as statements, loops, and all body functions).

Example: This code uses a new storage variable to correctly calculate the factorial of each callback.

JavaScript Code

```
function FactorialLet(n) {
if (n === 0) {
return 1;
} else {
let result = n *factorialLet(n - 1) ) );
// Let's create a new binding so that it appears again;
}

console.log(factorialLet(5)); // Output 120
```

Please use the code carefully.

3. const:

Block scope: Same as let but cannot be reallocated after initialization.

Example: This code tries to modify a const variable and results in an error.

JavaScript

```
function FactorialConst(n) {
if (n === 0) {
return 1;
}
else {
const result = n * FactorialConst(n - 1) ) );
// error: const cannot be returned
```

```
result ++;
// This line will cause the error to occur again;
}
console.log(factorialConst(5 ) );
// It will give an error. Use the code carefully.
```

Important:

In general, it is preferable to use let instead of var in recursion to prevent behavior in the variable dialog from being taken into account.

Use const variables in the following cases. The value must remain unchanged in the block to prevent incorrect changes.

Note that due to var's function-scoping nature, it has potential pitfalls, especially in recursion.

I hope these examples clarify the differences and help you write better code again!

JS Operators

Instanceof Operator

```
class testClass {
    constructor(name){
        this.name = name
    }
    getName(){
        console.log(`${this.name}`)
    }
}

let t1 = new testClass("John")
t1.getName()
console.log(t2 instanceof testClass)
```

In Operator

```
let obj = {
    name :  "John",
    age : "25",
    salary : "$2000"
}
console.log("name" in obj)
```

delete

```
delete obj.name
console.log(obj)
```

Find the largest among three

```
// Normal if-Else
const num1 = 10;
const num2 = 25;
const num3 = 15;
var largestNumber;

    if (num1 ⩾ num2 && num1 ⩾ num3) {
        largestNumber =  num1;
    } else if (num2 ⩾ num1 && num2 ⩾ num3) {
        largestNumber =  num2;
    } else {
        largestNumber =  num3;
    }



console.log("The largest number is:", largestNumber);
```

```
// nested if
const num1 = 10;
const num2 = 25;
const num3 = 15;
var largestNumber;

    if (num1 ⩾ num2) {
        if (num1 ⩾ num3) {
            largestNumber = num1;
        } else {
            largestNumber =  num3;
        }
    } else {
```

```
        if (num2 ⩾ num3) {
            largestNumber =  num2;
        } else {
            largestNumber =  num3;
        }
    }


console.log("The largest number is:", largestNumber);
```

Difference:

Readability: The && operator approach tends to be more concise and readable, especially when dealing with multiple conditions.

Structure: Nested if statements allow for more complex branching logic, as you can have different actions for different combinations of conditions.

Performance: In most cases, there won't be a significant difference in performance between the two approaches. However, the && operator might be slightly more efficient as it evaluates all conditions simultaneously and short-circuits if any condition is false, whereas nested if statements require sequential evaluation.

```
var num1 = +prompt("Enter first number : ");
var num2 = +prompt("Enter second number : ");

var choice = prompt("Enter your choice, 1. ADD , 2.SUB, 3. MUL,
4. DIV : ")

var result;

if(choice == 1){
    result = num1 + num2
    console.log(result)
}

else if( choice == 2){
    result = num1 - num2
    console.log(result)
}

else if(choice == 3){
    result = num1 * num2
    console.log(result)
}

else if( choice == 4){
    result = num1 / num2
    console.log(result)
}

else {
    console.log("Enter valid choice")
}
```