

# JAVASCRIPT

## ➤ **Introduction:**

- JavaScript is a high-level, Object-oriented, Multi-paradigm programming language.
- JavaScript allows us to add dynamic effects in pages.
- And it is also used for building entire web applications in the browser.

## ➤ **Inline JavaScript:**

- Inline JavaScript means the JavaScript written in HTML file.

## ➤ **External JavaScript:**

- There is separate JavaScript file with all logic having .js extension and attach in html file with script tag.
- This file make code modular and clear.
- Syntax: <script src="filename.js"></script>

## ➤ **Values and Variable:**

- Values are an information or data.
- Variable are block whose store this value or information or data.
- Variable name should be camelCase
- Variable only contains letters, numbers, underscore '\_', \$ sign.
- Variable should not start with numbers. Eg: 3name = 'hii'; X
- Variable may start with Underscore and dollar \$
- For constant variable write variable in UPPERCASE. Eg: PI = 3.14 ✓

## ➤ **Data Types:**

- JavaScript has a dynamic typing -> We do not have manually define the data type of the value stored in a variable. Instead, data types are determined automatically.
- To check data type of variable we use **typeof** operator. Eg: **typeof** name

### 1. **Primitive Data Types:**

- I. **Numbers:** Floating point numbers -> Used for decimals and integers.  
Eg: let age = 23;
- II. **String:** Sequence of characters -> Used for text. Eg: let name = "js";
- III. **Boolean:** Logical type that can only be true or false -> Used for decision taking. Eg: let isLogin = true;
- IV. **Undefined:** Value taken by a variable that is not yet defined ('empty value') Eg: let children;  
- If we only declare variable the value and type both are undefined.
- V. **Null:** Also means 'empty value'.

1. **NOTE:** Null in JavaScript has a **typeof object**, but it is a biggest bug in JavaScript language.

VI. **Symbol:** Value that is unique and cannot be changed.

VII. **BigInt:** Larger integer than the Number type can told.

## 2. Non – Primitive Data Types:

### ⊕ Dynamic typing:

```
let lastName = "Jangale";
console.log(typeof lastName); //String
lastName = 100;
console.log(typeof lastName); //number
```

- String can be converted to number dynamically.

### ➤ Let, const and var:

#### 1. Var:

- Var is oldest keyword in JavaScript for declaring a variable.
- It has a global scoped or function scope that means variable defined outside a function can be accessed globally and variable defined inside a particular function can be accessed within the function.
- Eg:

```
var a = 10
function f() {
    var b = 20
    console.log(a, b)
}
f();
console.log(a);
```

- We can re-declare a variable with same name in the same scope using var keyword, which give no error in case of var keyword

Eg:

```
var a = 10

var a = 8

a = 7
console.log(a);
```

## 2. Let:

- The let keyword is an improved version of the var keyword.
- It is introduced in ES6 or ECMAScript 2015.
- These variables have the block scope. It can't be accessible outside the particular code block.

Eg:

```
let a = 10;
function f() {
    let b = 9
    console.log(b);
    console.log(a);
}
f();
```

- Redeclaration in same scope of let variables is not allowed in JavaScript and it is the biggest advantages of let variables over var variables.

```
let a = 10

// It is not allowed
let a = 10

// It is allowed
a = 10
```

- But redeclaration in different scope is allowed in JavaScript.

Eg.

```

let a = 10
if (true) {
    let a = 9
    console.log(a) // It prints 9
}
console.log(a) // It prints 10

```

- We can modify the let value after declaring the value.

### 3. Const keyword:

- Const has all the properties that are the same as the let keyword, except the user cannot update it and must assign it with a value at the time of declaration.
- These variables also have the block scope.
- It is mainly used to create constant variables whose values cannot be changed once they are initialized with a value like value of PI.

Eg:

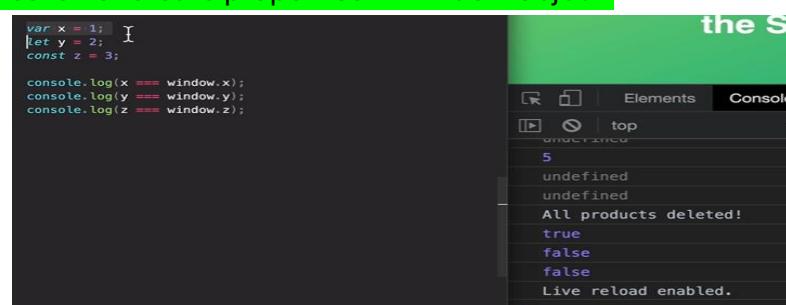
```

const a = 10;
function f() {
    a = 9
    console.log(a)
}
f();

```

**It gives TypeError: Assignment to constant variables.**

 **Note :** var variable creates properties in window object but let and const variables is not create properties in window object.



```

var x = 1; I
let y = 2; 
const z = 3;

console.log(x === window.x);
console.log(y === window.y);
console.log(z === window.z);

the S
Elements Console
top
5
undefined
undefined
All products deleted!
true
false
false
Live reload enabled.

```

➤ **Difference between let, var and const variables:**

<b>var</b>	<b>let</b>	<b>const</b>
Scope is functional or global	Scope is block scope only.	Scope is block scope only.
It can be updated and re-declared in the same scopes. (Mutable)	It can be updated but cannot be re-declared in the same scope. (Mutable)	It cannot be updated or re-declared in any scope. (Immutable)
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’	It cannot be accessed without initialization, as it cannot be declared without initialization.
These variables are hoisted.	These variables are hoisted but stay in the temporal dead zone until the initialization.	These variables are hoisted but stays in the temporal dead zone until the initialization.

⊕ **NOTE:** Don't use var keyword for declaring variable because var is outdated variable and it is not safe because it has a global scope. And there is no error occurs when we redeclare var variables with same name.

➤ **Global Variable:**

- If we declare variable without using var, let and const this variable consider as a global scope variable. We can access this variable from out off the scope and also inside the other scope.

**Eg:**

```
{
  a = 10;
}
console.log(a) //Out scope

{
  console.log(a) //Different scope
}
```

- But you should not create variable without var, let and const keyword until and unless any requirement available.

➤ **Template Literals:**

- If we want to insert variable in string, we use backticks (``) and pass variable in \${}.

- Template literals allowing us for multi-line string or string interpolation with embedded expressions.

Eg:

```
const firstName = "Lokesh";
const lastName = "Jangale";
const job = "Junior Software Engineer";
const company = "Smart Stream Technology";
const year = 2024;
|
const details = `Hii, My name is ${firstName} ${lastName}
I am ${year - 2002} years old
I work as ${job} in ${company}`;

console.log(details)
```

Output:

```
PS C:\Users\LJangale\Desktop\SST-Learning\JavaScript\01-Fundamentals-Part-1> node .\TemplateLiterals.js
Hii, My name is Lokesh Jangale
I am 22 years old
I work as Junior Software Engineer in Smart Stream Technology
```

## ➤ Type Conversion:

- JavaScript provide a way to convert data type manually.

### 1. String to Number:

- Syntax: Number(value)
- If we contain a string having number and we want to convert this string to number we use these syntax.

Eg:

```
let value = "23";
console.log(Number(value)) //output: 23→number
- If variable contain string as a word the conversion output is NaN
(Not a Number)
```

Eg:

```
let value = "Lokesh";
console.log(Number(value)) //output: NaN -> number
```

### 2. Number to String:

- Syntax: String(value);
- If we want to convert number to string then we use above syntax.

## ➤ Type coercion:

- Type coercion refers to the process of automatic or implicit conversion of value from one data type to another data type.

- This includes conversion from Number to String, String to Number, Boolean to number etc.
- For String if we use '+' then numberString is concatenated but if we use '-' then string is converted to number.

Eg:

<code>console.log(5 + '5')</code>	<code>PS C:\Users\1</code>
<code>console.log(5 - '5')</code>	<code>55</code>
<code>console.log('5' - 5)</code>	<code>0</code>
<code>console.log(5 * '5')</code>	<code>0</code>
<code>console.log(5 / '5')</code>	<code>25</code>
<code>console.log(5 ** '5')</code>	<code>1</code>
<code>console.log('5' * 5)</code>	<code>3125</code>
	<code>25</code>

#### ✚ Falsy value:

- In JavaScript there are 5 falsy values are available [ 0, undefined, '', NaN, null].
- These 5 falsy values are used for false conditions.

#### ➤ Equality Operator == vs ===:

##### 1. === or Strict Equality operator:

- It is checking value and also data type of value are equal or not.
- It does not perform type coercion.

Eg:

```
console.log(18 === 18) //true
console.log ('18' === 18) //false
```

##### 2. == or Loose Equality Operator:

- It is checking value but not check data type of value.
- It performs type coercion.
- That's why sometime bugs found

Eg:

```
console.log (18 == 18) //true
console.log ('18' == 18) //true
```

✚ **Note:** Always use strict equality operator (==) to check equality for value until and unless if any particular requirement are not present for ==.

#### ➤ Ternary Operator (?):

- It is a conditional operator which is used to check conditions true or false.
- It works as a single line if – else statement.

- Syntax:

**Condition ? true statement : false statement;**

- We call it a ternary operator because it has a tree main type condition, true statement and false statement.
- We can use ternary operators in template literals but we cannot use if-else statement in template literals.

Eg:

```
let age = 15;
console.log(`I like to drink ${age >= 18 ? "wine 🍷" : "water 💧"}`)
```

#### ➤ Strict Mode:

- Being a scripting language, sometimes the JavaScript code displays the code displays the correct result even it has some errors. Hence this is cause a bug in a program. To overcome this problem we can use the JavaScript strict mode.
- JavaScript provides “**use strict**”; expression to enable the strict mode. If there is any silent error or mistake in the code, it throws an error.
- We can write more safe code using strict mode.
- By using strict mode we can perform strict type checking in code. And it avoid global variable declaration
- The purpose of “**use strict**” is to indicate that the code should be executed in “strict mode”.
- With strict mode, you cannot use undeclared variables or we cannot declare variable without let, const and var.

Eg:

```
//Code without using strict mode
/*
let hasDrivingLicence = false
const passTest = true

if (passTest) hasDrivingLicence = true
if (hasDrivingLicence) console.log("I can drive now");
*/

//Above code is not giving error when i was assign value to wrong variable thats why it cause a bug in our program
'use strict'
let hasDrivingLicence = false;
const passTest = true;

if (passTest) hasDrivingLicence = true;
if (hasDrivingLicence) console.log("I can drive now");

//This code give us a error message about hasDrivingLicence variable is not defined
```

#### ➤ Function:

- Function is a piece of code which we can use repeatedly by just calling that function.

- We can also pass arguments to function, and function can also return some value.
- This is also called as function declaration.
- Functions allow us to write more maintainable code.

- Syntax:

```
function funcName(param){} //Function Declaration
```

```
funcName(val); //Function Calling
```

Eg:

```
//Normal Function
function calAge1(param) {
    return 2024 - param;
}
console.log(calAge1(2002))
```

## ➤ Types of Function:

### 1. Anonymous Function:

- Function without name is called Anonymous Function.
- This function is declaring as variable declaration
- This is also called as function expression.
- Syntax:

```
const funcName = function (param){}
```

Eg:

```
const calAge = function (param) {
    return 2024 - param;
}
const age = calAge(2002);
console.log(age);
```

 **Note:** The main difference between function declaration and function expression is in function declaration we can call function before declaration but in function expression we cannot call function before declaration.

**a) Function Declaration:**

```
//Function declaration  
console.log(calAge1(2002))  
function calAge1(param) {  
    return 2024 - param;  
}
```

Output:

```
PS C:\Users\LJangale\Desktop\SST-Learning>  
Hi I am lokesh  
5 0  
Juice of 5 apples and 0 oranges  
0 5  
Juice of 0 apples and 5 oranges  
22  
22
```

- This function can execute before declaration
- We cannot use function declaration for callback function.

**b) Function Expression:**

```
//Function expression  
const age = calAge(2002);  
console.log(age);  
const calAge = function (param) {  
    return 2024 - param;  
}
```

Output:

```
ReferenceError: Cannot access 'calAge' before initialization  
at Object.<anonymous> (C:\Users\LJangale\Desktop\SST-Learning\Java>  
at Module._compile (node:internal/modules/cjs/loader:1358:14)
```

- This function is not execute before declaration
- We can use function expression for callback function.

**2. Arrow Function:**

- It is concise way of writing JavaScript functions in shorter way.
- They make our code is more structured and readable.

- Arrow function is anonymous function that is function without a name but they are often assigned to any variable.
- They are also called as **Lambda function**.
- Syntax:  
`const funcName = () => {}`
- The return statement and function brackets are optional for single-line functions.

Eg:

```
//Arrow function

const yearsUntilRetirement = (birthYear, firstName) => { //Arrow Function declaration
  let age = 2024 - birthYear;
  const retirement = 60 - age;
  return `${firstName} is retires in ${retirement} years`;
}
console.log(yearsUntilRetirement(2002, "Lokesh")) //Arrow Function Calling
```

## ❖ All Function type in one image: ↗

### FUNCTIONS REVIEW: 3 DIFFERENT FUNCTION TYPES

#### 👉 Function declaration

Function that can be used before it's declared

```
function calcAge(birthYear) {
  return 2037 - birthYear;
}
```

#### 👉 Function expression

Essentially a function value stored in a variable

```
const calcAge = function (birthYear) {
  return 2037 - birthYear;
};
```

#### 👉 Arrow function

Great for a quick one-line functions. Has no this keyword (more later...)

```
const calcAge = birthYear => 2037 - birthYear;
```

👉 Three different ways of writing functions, but they all work in a similar way: receive input data, transform data, and then output data.

## ➤ Array:

- An array is a group of similar elements or a data item of the same type collected at contiguous memory locations.
- Syntax:

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

Or

```
const array_name = new Array(item1, item2, ...);
```

Eg:

```
const friends = ["Lokesh", "Yeshwant", "Vishal"];
console.log(friends);

//Another way to declare array by using new keyword
const year = new Array(2001, 2002, 2003, 2004);
console.log(year);

//Find the length of array
console.log(friends.length);

//Find the last element of array
console.log(friends[friends.length - 1])

//Change the last element of array
friends[friends.length - 1] = "Karan";
console.log(friends);
```

 **Note:** If array is declared as a **const** we can modify the element of array but we cannot assign new array to that variable

Eg:

```
friends[1] = "Vikram"; //allowed
console.log(friends)
friends = ["Lukky"] //not allowed
```

**Output:**

```
[ 'Lokesh', 'Yeshwant', 'Karan' ]
[ 'Lokesh', 'Vikram', 'Karan' ]
C:\Users\LJangale\Desktop\SST-Learning\JavaScript\
friends = ["Lukky"] //not allowed
^

TypeError: Assignment to constant variable.
```

#### ➤ **Object:**

- Object is a most important data type in JavaScript.
- Objects are store value as like array but in unique way.
- Array can store value in the form of index but object can store value is in the form of key-value pair.
- We can access element of array using index but we can access element of object by there keys.

- Object is like container in JavaScript which can hold multiple value in it.  
These value are store as a properties of object with its own key
- Syntax:

```
const object_name = {
    Key : value,
    ....
}
```

- We can access key by using dot (.) or square brackets (['Key'])

Eg:

```
console.log(lokesh.firstName);
console.log(lokesh['firstName'])
```

- The main difference between dot and square brackets is we can pass the expression in brackets, but we can't pass expression in dot.

Eg:

```
const nameKey = "Name";
console.log(lokesh."first" + nameKey) //Not allowed
console.log(lokesh['first' + nameKey]) //Allowed
```

- If we want to find value through any expression then only use bracket notation otherwise use dot notation as it gives more cleaner and simple code
- We can also pass function as a property in object

Eg:

```
//We can also pass function in object
const demo = {
    firstName: 'Lokesh',
    lastName: 'Jangale',
    birthYear: 2002,
    friends: ["Yashwant", "Vishal"],
    hasDrivingLicence: true,
    calcAge: function () {
        return now - this.birthYear;
    }
};

console.log(demo.calcAge())
```

 **Note:** **this** Represent current object.

➤ **Type of console. :**

1. **console.log();** -> Generate log value
2. **console.warn();** -> use to generate warning in console.
3. **console.error();** -> use to generate error in console.
4. **console.table();** -> log value display in table format.

Eg:

The screenshot shows a browser's developer tools console. At the top, there is a code block defining an object named 'demo' with properties like 'firstName', 'lastName', 'birthYear', 'friends', 'hasDrivingLicence', and a 'calcAge' method. Below the code, several console statements are shown: 'console.log(10)', 'console.warn(10)', 'console.error(10)', and 'console.table(demo)'. The output for 'console.log(10)' is '10' at VM179:1. The output for 'console.warn(10)' is 'VM179:2'. The output for 'console.error(10)' is 'VM179:3'. The output for 'console.table(demo)' is a table at VM179:4. The table has columns for '(index)', 'Value', '0', and '1'. The rows represent the properties of the 'demo' object: firstName ('Lokesh'), lastName ('Jangale'), birthyear (2002), friends ('Yashwant'), hasDrivingLicence (true), and an 'Object' row.

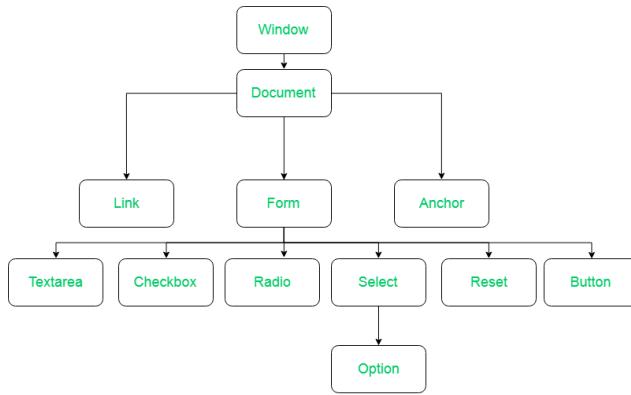
(index)	Value	0	1
firstName	'Lokesh'		
lastName	'Jangale'		
birthyear	2002		
friends		'Yashwant'	
hasDrivingLicence	true		'Vishal'
► Object			

➤ **Software Bug:**

- Defect or problem in a computer program.
- Basically, any unexpected or unintended behaviour of a computer program is a software bug.

➤ **Document Object Model (DOM):**

- DOM is a structured representation of HTML or XML documents in tree format.
- It allows JavaScript to access HTML elements and style to manipulate them.
- So basically, DOM is an API that represents and interacts with HTML or XML documents.
- We can also say that DOM is a connection point between JavaScript code and Html code.
- It is generated by browser when any HTML page is load in browser and it demonstrate HTML element in tree like structure.



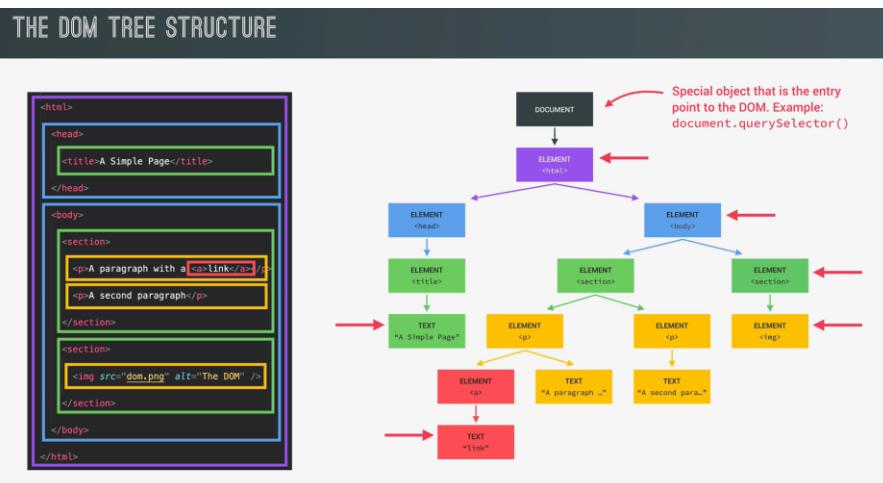
### ➤ **Window Object:**

- Window object is object of the browser which is always at top of the hierarchy. It is like an API that is used to set and access all the properties and method of the browser.
- It is automatically created by browser.

### - **Why is DOM Required?**

- HTML is used to structure the web pages and JavaScript is used to add behaviour to our web pages.
- When an HTML file loaded into the browser, the JavaScript cannot understand the Html document directly. So, it interprets and interact with DOM. Which is created by the browser based on the HTML documents.

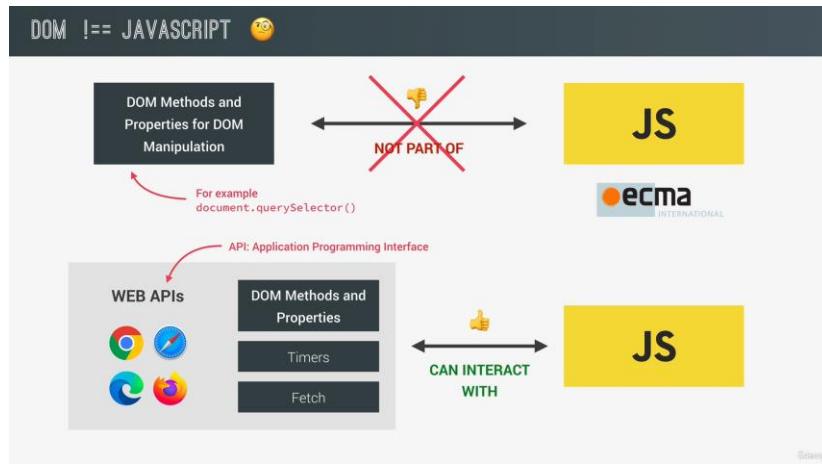
### - **The DOM Tree structure:**



- This type of structure is generated by browser

### ➤ **DOM!= JavaScript:**

- DOM is not a part of JavaScript language it is a part of WEB APIs.
- WEB APIs is a library that browsers implement which we can access using JavaScript code.
- That is a reason DOM manipulation is work a same in all browsers.



- There are various other APIs present in WEB APIs just like Times, Fetch etc.

## ➤ DOM Manipulation:

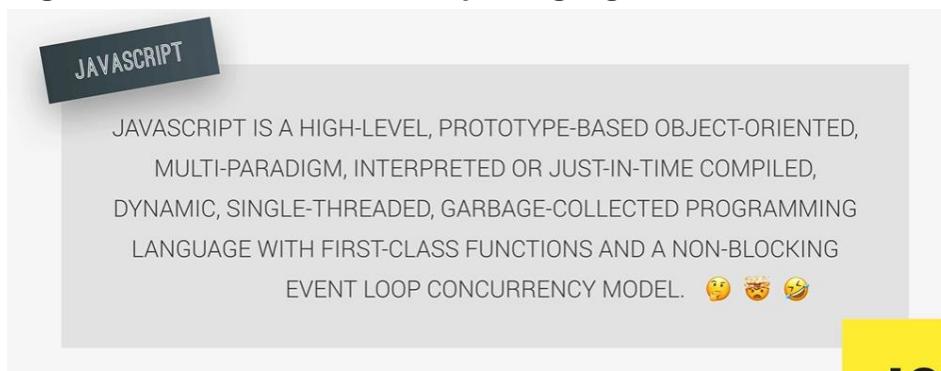
### 1) `document.querySelector("")`:

- It is used to select any HTML element by using ID, className, tagName etc.
- We can perform various manipulation using this like text printing, Modifying that text.

Eg:

```
> document.querySelector(".message").textContent
< 'Start guessing...'
> document.querySelector(".message").textContent = "Hello world"
< 'Hello world'
>
```

## ➤ High Level Overview of JavaScript Language:



### 1. High Level:

- There are certain low-level languages like C where we need to manually manage resources such as Memory, RAM etc.
- But in High-Level Language Developer does not have to worry about resources it manages automatically.

- It can make developer to write code easily, but disadvantages of high-level language are it is slower than low-level language like C.

## 2. Garbage Collector:

- All Stubs for memory management for JavaScript is done by GC.
- GC can remove unnecessary managed resource from memory and reclaim this memory.
- It works as a cleaner in JavaScript.

## 3. Interpreted or just-in-time compiler:

- JavaScript interpreter can compiler JavaScript code in Machine code.
- Because computer is not understanding JavaScript code directly, it only understands binary code.
- This can be done in JavaScript Engine.

## 4. Multi-paradigm:

- **Paradigm:** An approach and mindset of structuring code, which will direct your coding style and technique.
- There are 3 types of paradigm present in JavaScript.
  - a. Procedural Programming
  - b. Object-Oriented Programming (OOP)
  - c. Functional Programming (FP)

## 5. Prototype-based object-oriented:

- Almost everything in JavaScript is object except primitive value.

## 6. Dynamic:

- JavaScript is dynamically type language means when we declare variable we never mention datatype we only mention “**Let, const, var**”
- Type of variable becomes known at runtime based on value.
- If we create variable with string value and then in next line, we mention number in same variable then datatype is automatically convert from string to number.

## 7. First Class Function:

- In a language with first-class function, functions are simply treated as variables. We can pass them into other functions and return them from functions.

```
const closeModal = () => {
  modal.classList.add("hidden");
  overlay.classList.add("hidden");
};

overlay.addEventListener("click", closeModal);
```

Passing a function into another  
function as an argument:  
First-class functions!

## 8. Single-threaded:

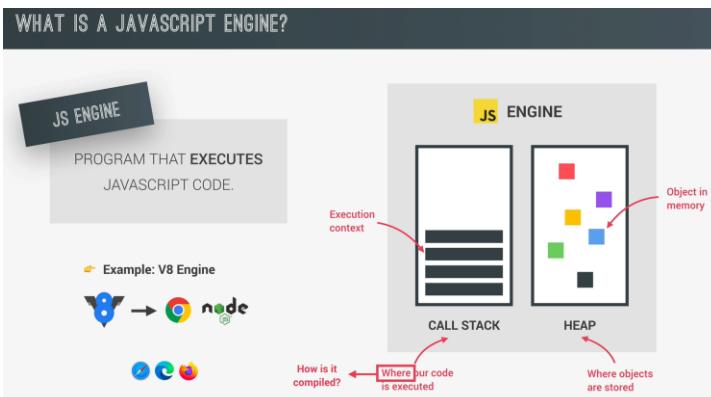
- **Concurrency model:** How the JavaScript engine handles multiple tasks happening at a same time.
- Because of JavaScript runs in on single thread, so it can only do one thing at a time.

## 9. Non-blocking event loop:

- JavaScript is single threaded language so it can't easily handle long-running task.
- We can achieve that using an **event loop:** Takes long running tasks, executes them in the “background”, and puts them back in the main thread once they are finished.

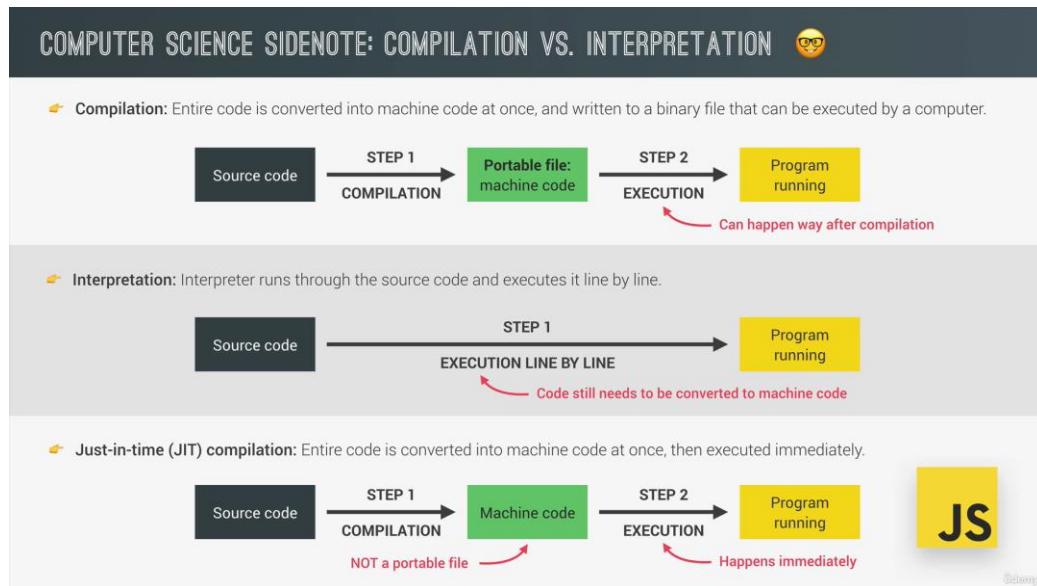
## ➤ JavaScript Engine:

- JavaScript Engine is simply a computer program that execute JavaScript code.
- JavaScript is a scripting language and is not directly understood by computer, but the browsers have inbuilt JavaScript engine which help them to understand and interpret JavaScript codes.
- These engine help to convert out JavaScript program into computer-understandable language.
- Every browser contains separate JavaScript engine but the most famous engine in **Google v8 Engine.** It is run on chrome as well as node js.

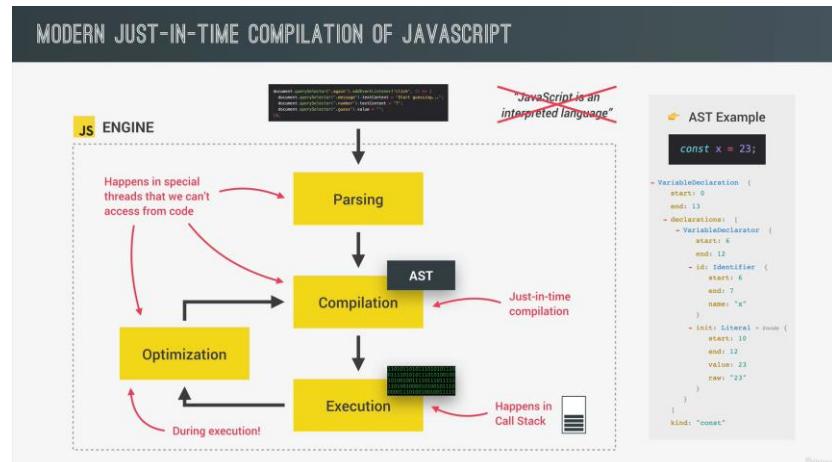


- **Call Stack:** Where our code is executed stack wise.
- **Heap:** Where all objects are stored.

- **Compilation vs Interpretation:**



1. We can say that old JavaScript are interpreted language, but interpretation makes JavaScript slower than other language
2. So, our modern JavaScript is **Just-in-time (JIT) compilation**.

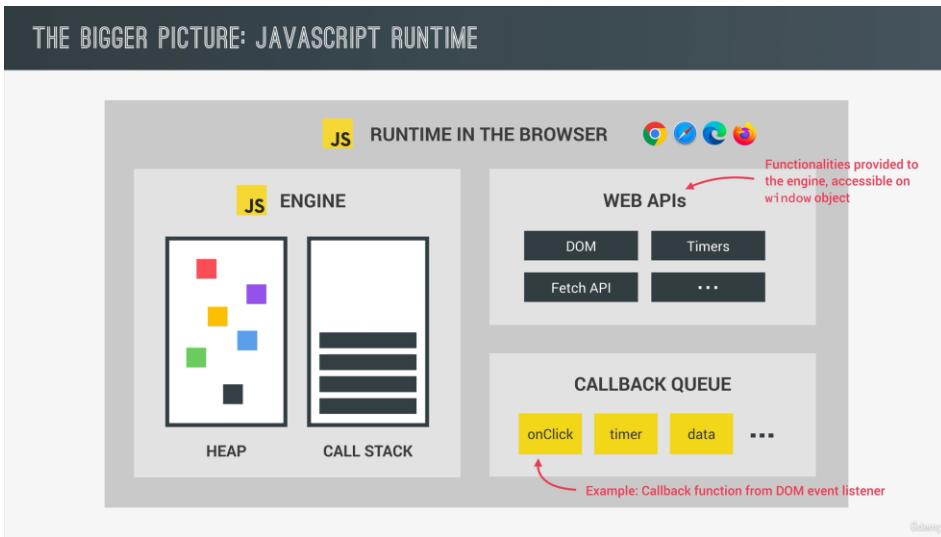


### 3. AST: Abstract Syntax Tree.

- **Different JavaScript Engines:**

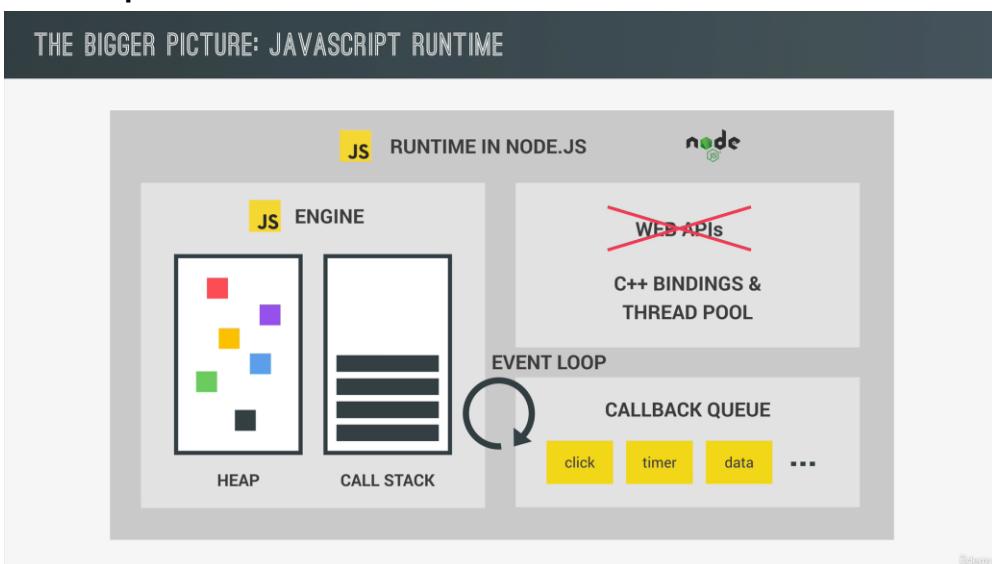
Browser	Name of Javascript Engine
Google Chrome	V8
Edge (Internet Explorer)	Chakra
Mozilla Firefox	Spider Monkey
Safari	Javascript Core Webkit

## ➤ JavaScript Runtime on Web Browser:



- All callback function is present in **callback queue**. When any event occurs or want to execute, function is loaded in **call stack** by using **event loop**.
- This process is executed in loop.

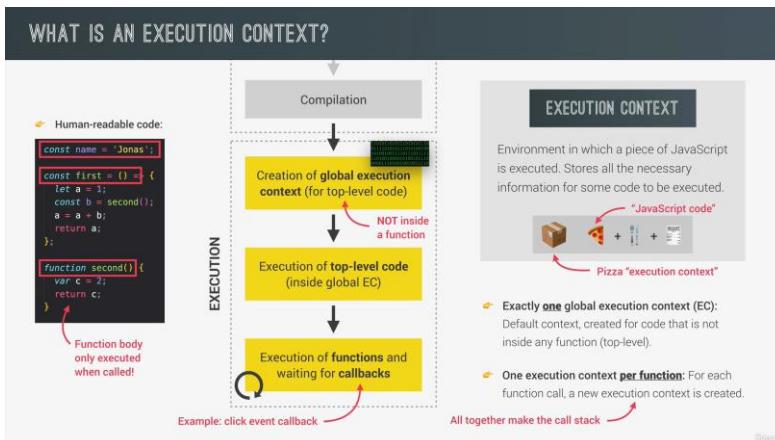
## ➤ JavaScript Runtime on Node/Outside the browser:



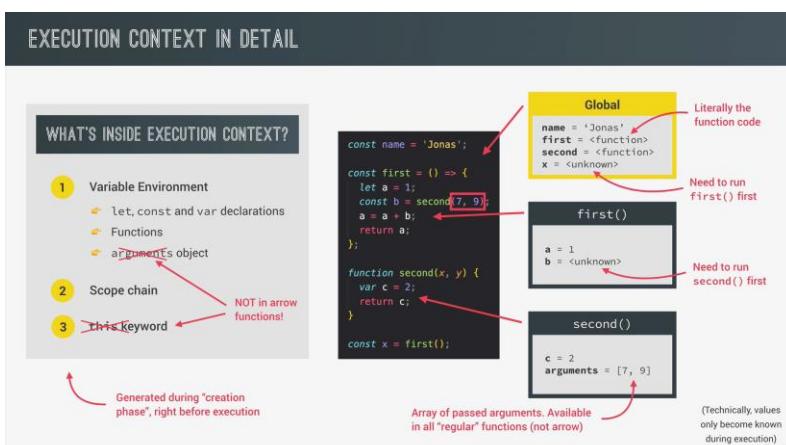
- As node is outside browser environment so **web APIs** are not available for node runtime instead of this **C++ Bindings and Thread Pool** are available.
- Execution of **callback event** are same as browser runtime environment.

## ➤ Execution Context:

- When the JavaScript engine scans a script file, it makes an environment called the **Execution Context** that handle the entire transformation and execution of the code.
- During the context runtime, the parser parse the source code and allocates memory for the variables and functions. The source code is generated and gets executed.
- There are two types of execution context : **global** and **function**.
- The **global** execution context is created when a JavaScript first starts to run, and it represent global scope.
- A **function** execution context is created whenever a function is called, representing the function's local scope.



## ❖ Execution context in detail:



## ❖ Phases of the JavaScript Execution Context:

**1. Creation Phase:** In this phase, The JavaScript engine creates the execution context and sets up the script's environment. It determines the value of variables and functions and sets up the scope chain for the execution context.

MEMORY	CODE
Variable : undefined Function : {...}	Each line of source code is executed line by line from top to bottom.

- In creation phase only variable is created without, but value is not assign. The default value is **undefined**.
- For function also same work only function name is defined like 

MEMORY	CODE
n : undefined square : {...} square1 : undefined square2 : undefined	

## 2. Execution Phase:

- In this phase, the JavaScript engine executes the code in the execution context. It processes any statements or expressions in the script and evaluates any functions calls.
- In this phase, it starts going through entire code line by line from top to bottom. And assign the value to every variable which define in creation phase. Until now, the value of variables was undefined by default.
- For every function invoke in execution phase new function execution context is created where both creation and execution phase happen. 

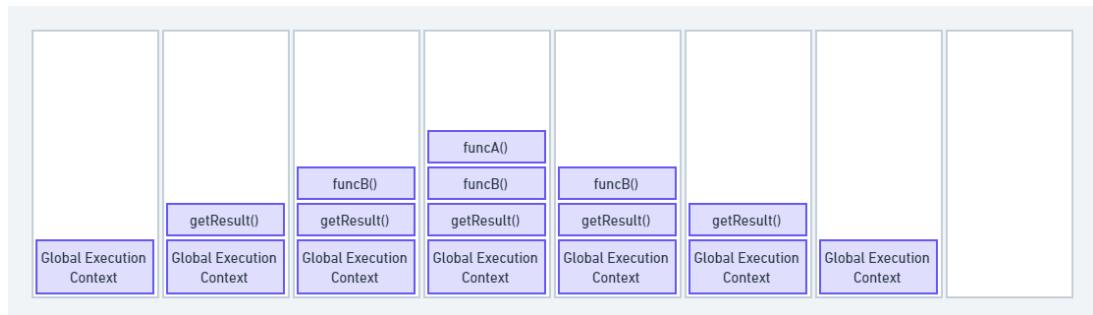
MEMORY	CODE				
n : 5 square : {...} square1 : undefined square2 : undefined	<table border="1"> <thead> <tr> <th>MEMORY</th><th>CODE</th></tr> </thead> <tbody> <tr> <td>n : undefined ans : undefined</td><td></td></tr> </tbody> </table>	MEMORY	CODE	n : undefined ans : undefined	
MEMORY	CODE				
n : undefined ans : undefined					

- Then function return the value and **function execution context** will be destroyed.

MEMORY	CODE
<pre>n : 5 square : {...} square1 : 25 square2 : 64</pre>	

➤ **Call Stack:**

- For call stack description click on [call stack](#).
- To keep the track of all the contexts, including global and functional, the JavaScript engine used **call stack**.
- A call stack also known as an “**Execution Context Stack**”, “**Runtime Stack**” or “**Machine Stack**”.
- It uses the **LIFO** principle. When the engine first starts executing the script, it creates global context and pushes it on the stack.
- Whenever function is invoked, similarly, the JS engine crate function stack context for the function and pushes it to the top of the call stack and starts executing it.
- When the execution of current function is complete, then the JavaScript engine will automatically remove the context from the call stack, and it goes back to its parent.



⚠ **Note:** JavaScript only work on single thread that's why only one function can be executed at a time in call stack.

➤ **Scoping and Scope Chain in JavaScript:**

- **Scoping:** How our program's variables are organized and accessed. “Where do variables live?” or “Where can we access a certain variable, and where not?”.
- **Lexical Scoping:** Scoping is controlled by placement of functions and blocks in the code.

- **Scope:** Space or environment or place in which a certain variable is declared (variable environment in case of functions). There is **global** scope, **function** scope and **block** scope.
- **Scope of a variables:** Region of our code where a certain variable can be accessed.

#### ❖ The 3 types of scope:

##### 1. Global Scope:

- Variable declare **outside** of any function or block is called global scope.
- Variable declared in global scope are accessible everywhere.

##### 2. Function Scope:

- Variables are accessible only **inside function, Not Outside** function.
- It is also called local scope.
- If we try to access variable **outside** function which declared inside in function, we can get **Reference Error**.

##### 3. Block Scope:

- Block scope introduce in **ES6**.
- Variables are accessible only **inside block scope or Curly braces {}**.
- However, this only applies to **let** and **const** variables. It is not applying on **var** variables.
- In **strict mode** function are also block scoped.

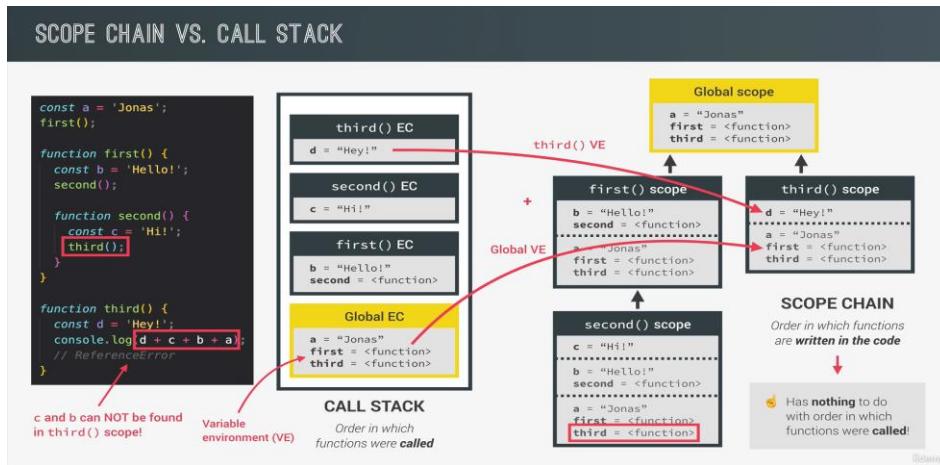
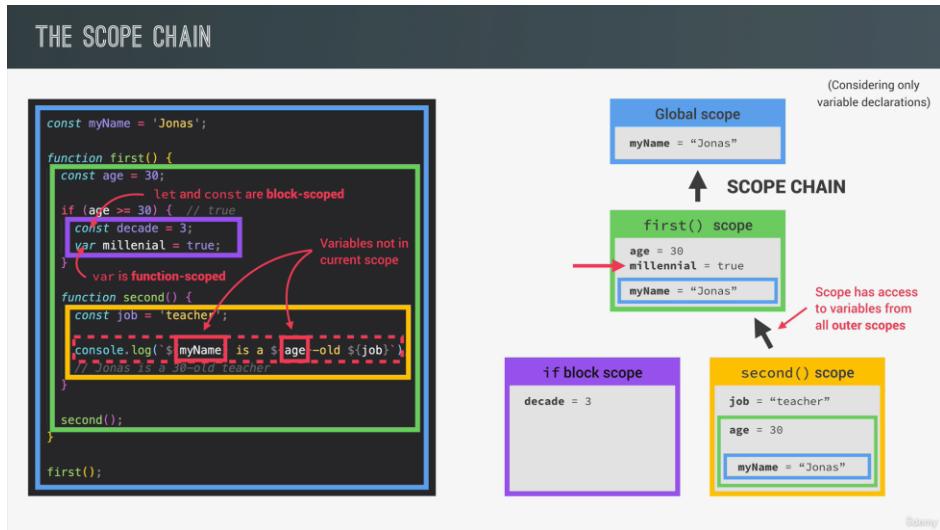
#### THE 3 TYPES OF SCOPE

GLOBAL SCOPE	FUNCTION SCOPE	BLOCK SCOPE (ES6)
<pre>const me = 'Jonas'; const job = 'teacher'; const year = 1989;</pre> <p>👉 Outside of <b>any</b> function or block 👉 Variables declared in global scope are accessible <b>everywhere</b></p>	<pre>function calcAge(birthYear) {   const now = 2037;   const age = now - birthYear;   return age; }  console.log(now); // ReferenceError</pre> <p>👉 Variables are accessible only <b>inside function, NOT outside</b> 👉 Also called local scope</p>	<pre>if (year &gt;= 1981 &amp;&amp; year &lt;= 1996) {   const millenial = true;   const food = 'Avocado toast'; } // Example: if block, for loop block, etc.  console.log(millenial); // ReferenceError</pre> <p>👉 Variables are accessible only <b>inside block</b> (block scoped) ⚠️ <b>HOWEVER</b>, this only applies to <b>let</b> and <b>const</b> variables! 👉 Functions are <b>also block scoped</b> (only in strict mode)</p>

⊕ **Note:** **let** and **const** are block scope but **var** is function scope. We can access **var** variable outside the block scope but we can't access **var** variable outside function.

- **Function** is a block scope only in **strict mode**. If we remove **use strict** then we can access function outside a block also.

### ❖ The Scope chain:



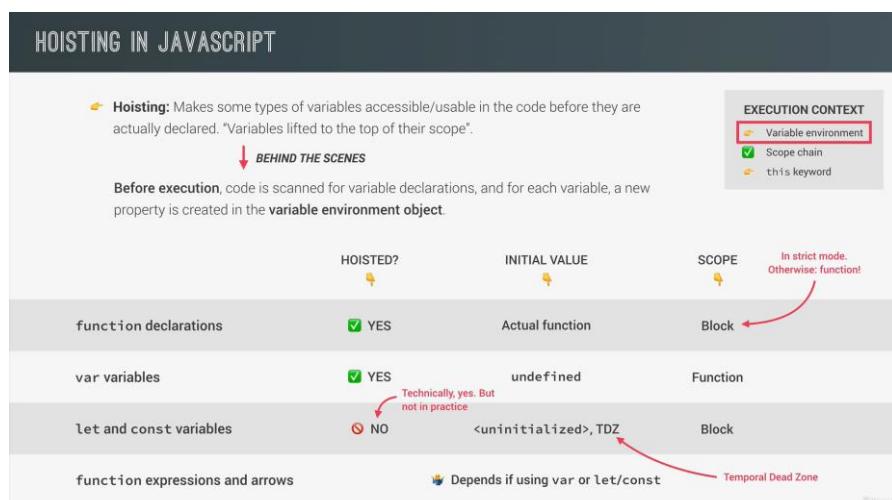
### ❖ Scoping Summary:

- Scoping asks the question “Where do variables live?” or “Where can we access a certain variable, and where not?”.
- There are 3 types of scope in JavaScript: **global scope**, **function scope** and **block scope**.
- Only **let** and **const** variables are block-scoped. Variables declared with **var** end up in the closest function scope.
- In JavaScript, we have lexical scoping, so the rules of where we can access variables are based on exactly where in the code functions and blocks are written.

- Every scope always has access to all the variables from all its outer scopes. This is the scope chain!
- When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it's looking for. This is called variable lookup.
- The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope.
- The scope chain in certain scope is equal to adding together all the variable environments of the all-parent scopes.
- The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all!

## ➤ Hoisting in JavaScript:

- Makes some types of variables accessible/usable in the code before they are actually declared. "Variables lifted to the top of their scope".
- **Before execution**, code is scanned for variable declarations, and for each variable, a new property is created in the **variable environment object**.



- Hoisting is the default behaviour in JavaScript where declarations of variables and functions are moved to the top of their respective scopes during the compilation phase.
- This means regardless of where variables and functions are declared within a scope, they are accessible throughout that scope.
- JavaScript only hoists declarations, not initializations.
- Hoisting allows calling a function before its declaration.

## ❖ Temporal Dead Zone (TDZ):

- For a particular variable, the Temporal Dead Zone is the area starting from the beginning of the current scope to that variable's declaration.
- TDZ are used for: **It makes easier to avoid and catch errors which is accessing variable before declaration is bad practice and it should be avoided.**

- It also makes **const** variable actually work, we know that **const** variable have one time declaration and initialization property so if we use **const** before declaring or initializing then it considers as a bad practice. That's why **TDZ** help us to avoid this mistake.

### ❖ Why Hoisting?

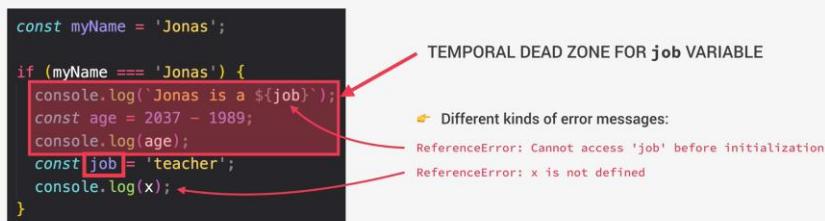
- Developer adds hoisting functionality in JavaScript to use function declaration before they are actual define.

```
// Function declaration
sayHello();

function sayHello() {
  console.log("Hello!");
}
```

- But it only works with function declaration type it can't work with function expression or arrow function.

## TEMPORAL DEAD ZONE, LET AND CONST



The screenshot shows a snippet of JavaScript code with a red box highlighting the assignment of the 'job' variable. A callout points to this box with the text "TEMPORAL DEAD ZONE FOR job VARIABLE". Another callout points to the 'job' variable with the text "Different kinds of error messages: ReferenceError: Cannot access 'job' before initialization ReferenceError: x is not defined".

```
const myName = 'Jonas';
if (myName === 'Jonas') {
  console.log(`Jonas is a ${job}`);
  const age = 2037 - 1989;
  console.log(age);
  const job = 'teacher';
  console.log(x);
}
```

### WHY HOISTING?

- ↳ Using functions before actual declaration;
- ↳ var hoisting is just a byproduct.

### WHY TDZ?

- ↳ Makes it easier to avoid and catch errors: accessing variables before declaration is bad practice and should be avoided;
- ↳ Makes **const** variables actually work

### ☞ Hoisting for variable:

- **var** variable can access before declaration but it gives **undefined** value.
- **let** and **const** variable cannot access before declaration it gives **ReferenceError**.

Eg:

```
=====Hoisting for variables=====
console.log(myName); //undefined
console.log(age); //ReferenceError: Cannot access 'age' before initialization
console.log(gender); //ReferenceError: Cannot access 'gender' before initialization

var myName = "lokesh";
let age = 23;
const gender = "Male";
```

### ☞ Hoisting for function:

- For function declaration type hoisting of function is work fine.
- But for function expression type or arrow function hoisting is not work it give **ReferenceError**.

Eg:

```
//=====Hoisting for Function=====
console.log(addDecl(2, 3)); //5
console.log(addExpr(2, 3)); //ReferenceError: Cannot access 'addExpr' before initialization
console.log(addArrow(2, 3)); //ReferenceError: Cannot access 'addArrow' before initialization

function addDecl(a, b) {return a + b;}
const addExpr = function (a, b) { return a + b; }

const addArrow = (a, b) => a + b;
```

- But if we create function with **var** variable it will give **TypeError**

Eg:

```
//=====Hoisting for Function=====
console.log(addDecl(2, 3)); //5
console.log(addExpr(2, 3)); //TypeError: addExpr is not a function
console.log(addArrow(2, 3)); //ReferenceError: Cannot access 'addArrow' before initialization

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

var addExpr = function (a, b) { return a + b; }

const addArrow = (a, b) => a + b;
```

#### ☞ Hoisting with scope:

```
const x = 2;
{
  console.log(x); //ReferenceError: Cannot access 'x' before initialization
  let x = 3;
}
```

- It can first check **x** variable in current scope then if found but it is declared after calling that variable that's why it gives ReferenceError.

 **Note:** Always remember that in the background the JavaScript is first declaring the variable and then initializing them. It is also good to know that variable declaration are processed before any code executed.

#### ➤ ‘this’ keyword/variable:

- It is a special variable that is created for every execution context (every function).
- It takes the value of the ‘owner’ of the function in which the keyword is used.
- **this** is not static. It depends on how the function is called, and its value is only assigned when the function is actually called.
- **this** keyword refers to the current context or scope or object within which code is executing.
- **Arrow function** doesn’t get its own **this** variable. Although it uses **this** variable from there surrounding function (lexical this).

- Without strict mode **this** keyword in function refers global **window** function and contain all the properties of window function. But with strict mode **this** keyword in function give result as undefined.
- But for **arrow function** it refers surrounding function or window function with or without strict mode.

## HOW THE THIS KEYWORD WORKS

**this keyword/variable:** Special variable that is created for every execution context (every function). Takes the value of (points to) the "owner" of the function in which the **this** keyword is used.

**this is NOT static.** It depends on **how** the function is called, and its value is only assigned when the function is actually called.

**EXECUTION CONTEXT**

- Variable environment
- Scope chain
- this keyword**

**Method example:**

```
const jonas = {
  name: 'Jonas',
  year: 1989,
  calcAge: function() {
    return 2037 - this.year
  }
};
jonas.calcAge(); // 48
```

calcAge is method      jonas      1989  
Way better than using jonas.year!

- In object **this** variable refers to the current object. If we create a function in object, in that function if we want to access property of that object then we use **this** keyword which is denote that this is a property of current object.

Eg:

```
const lokesh = {
  name: "lokesh",
  year: 2002,
  calcAge: function () {
    console.log(2024 - this.year);
  }
}
```

By using **this.year** we can access year property of Lokesh object.

### ➤ Regular function vs Arrow function regarding this keyword:

- Arrow function does not contain **this** keyword. It inherits **this** from its parent function.

```
const lokesh = {
  firstName: 'Lokesh',
  year: 2002,
  calcAge: () => [
    console.log(this); // {}
    console.log(2024 - this.year); //NaN
  ]
}

lokesh.calcAge();
```

- Parent function of **calcAge** function is window function and window function does not contain any **year** property.
- That's why use of arrow function as a function is bad practice instead of use any regular function like function expression or function declaration.

```

const lokesh = {
  firstName: 'Lokesh',
  year: 2002,
  calcAge: function () {
    console.log(this); // { firstName: 'Lokesh', year: 2002, calcAge: [Function: calcAge] }
    console.log(2024 - this.year); // 22
  }
}

lokesh.calcAge();

```

- By using regular function, we can use all property of that object because regular functions contain **this** keyword.
- So, where we use arrow function let's see below example,

```

const lokesh = {
  firstName: 'Lokesh',
  year: 2002,
  calcAge: function () {
    console.log(this); // { firstName: 'Lokesh', year: 2002, calcAge: [Function: calcAge] }
    console.log(2024 - this.year); // 22
    const isEligible = function () {
      console.log(this); // undefined
      console.log((2024 - this.year) > 18); // TypeError: Cannot read properties of undefined
    }
    isEligible();
  }
}

lokesh.calcAge();

```

- As we saw **this** keyword is undefined because **isEligible()** function call is a just regular function call and the rule says that for every regular function call **this** keyword is set to undefined.
- To make this workable we have two options:

- Create a variable, assign it with this, and use it as this keyword

Eg.

```

const lokesh = {
  firstName: 'Lokesh',
  year: 2002,
  calcAge: function () {
    console.log(this); // { firstName: 'Lokesh', year: 2002, calcAge: [Function: calcAge] }
    console.log(2024 - this.year); // 22
    const self = this;
    const isEligible = function () {
      console.log(self); // { firstName: 'Lokesh', year: 2002, calcAge: [Function: calcAge] }
      console.log((2024 - self.year) > 18); // true
    }
    isEligible();
  }
}

lokesh.calcAge();

```

- Use arrow function instead regular function, because arrow function inherits **this** keyword from parent function so it can access all property of that function.

Eg.

```

const lokesh = {
  firstName: 'Lokesh',
  year: 2002,
  calcAge: function () {
    console.log(this); // { firstName: 'Lokesh', year: 2002, calcAge: [Function: calcAge] }
    console.log(2024 - this.year); // 22

    const isEligible = () => {
      console.log(this); // { firstName: 'Lokesh', year: 2002, calcAge: [Function: calcAge] }
      console.log((2024 - this.year) > 18); // true
    }
    isEligible();
  }
}

lokesh.calcAge();

```

- Regular functions have access of **arguments keyword**, but arrow function does not have access of **arguments** keyword.

## ➤ Reference vs Primitive value:

**PRIMITIVE VS. REFERENCE VALUES**

👉 Primitive values example:

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

👉 Reference values example:

```
const me = {
  name: 'Jonas',
  age: 30
};  
No problem, because  
we're NOT changing the  
value at address 0003!  
const friend = me;  
friend.age = 27;  
  
console.log('Friend:', friend);
// { name: 'Jonas', age: 27 }
  
console.log('Me:', me);
// { name: 'Jonas', age: 27 }
```

Identifier	Address	Value
age	0001	30
oldAge	0002	31
me	0003	D30F
friend		

CALL STACK

Address	Value
D30F	{ name: 'Jonas'; age: 27 }

HEAP

Reference to memory address in Heap

## ➤ Data Structure and Modern Operators:

### 1. Array:

#### ○ Destructuring of array:

- We can do these two ways:

```
//Destructuring of array
const arr = [1, 2, 3];
const a = arr[0];
const b = arr[1];
const c = arr[2];

console.log(a, b, c); //a=1,b=2,c=3

const [x, y, z] = arr;
console.log(x, y, z); //x=1,y=2,z=3
```

- If we want to add third value of array to second element, we just pass nothing and separate it by comma.

```
const [first, , second] = restaurant.categories;
console.log(first, second); //O/P: Italian Vegetarian
```

- **Swapping/Switching a variable in array without using third variable:**

```
[first, second] = [second, first]; //---> Swapping without
third variable
console.log(first, second); //o/p: Vegetarian Italian
```

- **Receive two return value from a function:**

```
const [starter, mainCourse] = restaurant.order(2, 0);
console.log(starter, mainCourse); // Garlic Bread Pizza
```

- **Destructuring nested array:**

```
//Destructuring nested array
const nested = [2, 4, [5, 6]];
const [i, , [j, k]] = nested;
console.log(i, j, k); //2,5,6
```

- **Swap value of 2 variable using array destructuring.**

```
//Swapping value of 2 variable using array destruturing
let o = 12;
let m = 13;
[m, o] = [o, m];
console.log(o, m);
```

### 3. Object:

- **Destructuring object:**

- For destructuring object, we need to mention the variable name as same as name of property in object
- In object order is not important so if we want to skip some element then we don't need to mention blank element instead we just mention required property.

```
//Destructuring the object
const { name, openingHours, categories } = restaurant;
console.log(name, openingHours, categories);
```

#### O/P:

```
Classico Italiano {
  thu: { open: 12, close: 22 },
  fri: { open: 11, close: 23 },
  sal: { open: 0, close: 24 }
} [ 'Italian', 'Pizzeria', 'Vegetarian', 'Organic' ]
```

- If want to give our own name to the variable, we use below syntax:

### Property : variableName

```
const {
  name: restaurantName, //Classico Italiano
  openingHours: hours, //{thu: { open: 12, close: 22 },    fri: { open: 12, close: 22 },
  categories: tags, // [ 'Italian', 'Pizzeria', 'Vegetarian', 'Organic' ]
} = restaurant;
console.log(restaurantName, hours, tags);
```

- For Default value:

```
const { menu = [], starterMenu: starters = [] } = restaurant;
console.log(menu, starters); //[] [ 'Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad' ]
```

Without default value we can get **undefined** if property is not present in object.

- Mutating Variables/Alter variables:

- If we want to mutate existing variable value with object variable value
- Syntax:

### ({existing\_variables = object})

```
//Mutating variables
let a = 111;
let b = 999;
const obj = { a: 23, b: 7, c: 14 };
({ a, b } = obj);
console.log(a, b); //a=23,b=7
```

- Nested Object:

```
//Nested Objects
const {
  fri: { open, close },
} = restaurant.openingHours;
console.log(open, close); //11 23
```

## ➤ Spread Operator(...):

- The spread operator helps us **expand an iterable** such as an array where multiple arguments are needed, it also helps to expand the object expressions. In cases where we require all the elements of an iterable or object to help us achieve a task, we use as spread operator.
- Spread Operator is a key feature in JavaScript that enables an iterable to expand whenever zero or more arguments are required.
- Its primary use case is with arrays, especially when expecting multiple values.

- This operator provides the convenience of easily extracting a list of parameters from an array, making our code more versatile and readable.
- **Syntax:**

```
let variableName = [...iterable];
```

Eg:

```
// spread operator doing the concat job or merge element
let arr = [1, 2, 3];
let arr2 = [4, 5];

arr = [...arr, ...arr2];
console.log(arr); // [ 1, 2, 3, 4, 5 ]
```

- Whenever we need a individual element in array we use spread operator(...).

Eg:

```
console.log(...arr); //1 2 3 4 5
```

- Difference between destructing array and spread operator is spread operator can takes all the element from array and it doesn't create new variable, however in destructing of array we can takes only required element in array and we can also store that element in variables.

- **Spread operator with string**

```
//working with string
const str = 'Lokesh';
const letter = [...str, ' ', 'J.'];
console.log(letter); //['L', 'o', 'k','e', 's', 'h', ' ', 'J.']
```

Spread operator does not work with template literals because for template literals only one value is required one arguments but spread operator returns multiple value.

```
console.log(` ${...str} `);
```

- **Working with function:**

```
//Working with function
function display(name1, name2, name3) {
  console.log(name1, name2, name3);
}
const nameArr = ['lokesh', 'Yashwant', 'Vikram'];
display(...nameArr); //lokesh Yashwant Vikram
```

- **Working with Objects:**

```
//Objects
const newRestaurant = { foundedIn: 1998, ...restaurant,
founder: 'Lokesh' };
console.log(newRestaurant);
```

- **Shallow copy of object**

```
//Shallow copy of object
const restaurantCopy = { ...restaurant };
restaurantCopy.name = 'Ristorante Roma';
console.log(restaurantCopy.name); //Ristorante Roma
console.log(restaurant.name); //Classico Italiano
```

- **Rest Parameter(...parameterName):**

- Rest parameter is **converse** to the spread operator.
- While spread operator expands of an iterable, the rest parameter compresses them. It collects several elements.
- Rest parameter is an improved way to handle function parameters, allowing us to more easily handle various input as parameters in a function.
- The rest parameter syntax allows us to represent an indefinite number of arguments as an array.
- With the help of rest parameter, a function can be called with any number of arguments, no matter how it was defined.
- It is also used to collect multiple elements and composed it in array

```
//REST, because on LEFT side of =
const [a, b, ...others] = [1, 2, 3, 4, 5];
console.log(a, b, others); //1 2 [ 3, 4, 5 ]
```

- **Rest operator** is use left side of assignment (=) operator. However, **spread operator** are use right side of assignment (=) operator.

```
//SPREAD OPERATOR
const arr = [1, 2, ...[3, 4]];
console.log(arr); // [ 1, 2, 3, 4 ]
```

```
//REST, because on LEFT side of =
const [a, b, ...others] = [1, 2, 3, 4, 5];
console.log(a, b, others); //1 2 [ 3, 4, 5 ]
```

- Rest parameter should always last parameter.

```
const [a, b, ...others,e] = [1, 2, 3, 4, 5]; //SyntaxError: Rest element must be last element
console.log(a, b, others); //1 2 [ 3, 4, 5 ]
```

- **Working with function:**

```
//function
const display = function (...numbers) {
  console.log(numbers);
};
display(2, 3); // [2, 3]
display(2, 3, 4, 5); // [2, 3, 4, 5]
```

- **Working with objects:**

```
//Objects
const { sat, ...weekDays } = restaurant.openingHours;
console.log(weekDays);

// [3, 4, 5]
{ thu: { open: 12, close: 22 }, fri: { open: 11, close: 23 } }
```

- **Short-circuiting:**

- Short-circuiting is a behaviour exhibited by logical operator (`&&`, `||`) where the evaluation of the second operand is skipped if the outcome can be determined by evaluating the first operand alone.
- There are 3 different properties of logical operator:
  - It uses ANY Data Type
  - It returns ANY Data Type
  - Short-circuiting

```
console.log(3 || 'lokesh'); // 3
console.log('' || 'Lokesh'); // Lokesh
console.log(true || 0); // true
console.log(undefined || null); // null
console.log(undefined || 0 || '' || 'Hello' || 23 || null); // Hello
```

- **Returning default value in multiple way**

If we want to return some value in case not any value is present, we can do like this.

```

//return default value if no value present in object using
ternary and short-circuit
//using ternary operator
restaurant.newGuest = 100;
const guest1 = restaurant.newGuest ? restaurant.newGuest :
10;
console.log(guest1); //10 --> if no value present then it
return 10 otherwise return value of restaurant.newGuest

//using short-circuiting
const guest2 = restaurant.newGuest || 20;
console.log(guest2);

```

But it is not work with falsy value mean If `restaurant.newGuest = 0` then it return default value

To Fix this in ES2020 new concept is introduces **Nullish Value**

#### ❖ Nullish Value:

- Nullish value are **null** and **undefined** only it is not contain other falsy value like **0**, **''**, **NaN**.
- It is fix that error

```

restaurant.newGuest = 0;
const guestCorrect = restaurant.newGuest ?? 20;
console.log(guestCorrect); //0

```

**Note:** **Or** operator return first truthy value or return last falsy value and **And** operator return first falsy value or return last truthy value.

```

console.log('--- AND ---');
console.log(0 && 'Lokesh');//0
console.log(7 && 'Lokesh');//lokesh --> return last truthy
value
console.log('Hello' && 23 && null && 'lokesh'); //null

```

#### - **Looping:**

##### - **For-of loop:**

- For-of is used to iterate over the array or other iterables.

**Syntax:**

```
for( const/let variable of iterable) {};
```

**Eg:**

```

//For-of loop
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];
for (const item of menu) {
  console.log(item);
}

```

- If we want to print value with their index use **.entries()** method

```
//If we want to print item with index value use .entries() method
for (const item of menu.entries()) {
  console.log(item);
}
```

```
[ 0, 'Focaccia' ]
[ 1, 'Bruschetta' ]
[ 2, 'Garlic Bread' ]
[ 3, 'Caprese Salad' ]
```

- Array indexing with different ways

```
//Print value with index using array indexing
console.log('-----Array indexing-----');
for (const item of menu.entries()) {
  console.log(` ${item[0]}: ${item[1]}`);
}

//Print value with index using array destructuring
console.log('-----Array Destructuring-----');
for (const [i, item] of menu.entries()) {
  console.log(` ${i}: ${item}`);
}
```

- **Optional Chaining(?.):**

- The optional chaining is an error-proof way to access nested object properties, even if an intermediate property doesn't exist.
- It is similar to **chaining ?** Except that it does not report the error, instead it returns a value which is **undefined**.
- It also works with a function call when we try to make a call to a method which may not exist.
- When we want to check a value of the property which is deep inside a tree-like structure, we often must perform check whether intermediate nodes exist.

```
//Optional Chaining
// console.log(restaurantNewUi.openingHours.mon.open); //TypeError: Cannot read property 'open'
console.log(restaurantNewUi.openingHours.mon?.open); //undefined

const days = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
for (const day of days) {
  const open = restaurantNewUi.openingHours[day]?.open ?? 'Closed';
  console.log(`On ${day} we open at ${open}`);
}
```

- It is also work on method, it checks method exists or not if exist then it execute that method or return undefined.

```
//Optional chaining on methods checking method exist or not
console.log(restaurantNewUi.order?.(0, 1) ?? 'Method does
not exists');
console.log(restaurantNewUi.orderRissotto?.(0, 1) ?? 
'Method does not exists'); //undefined or 'Method not
exist'
```

- Optional Chaining with array

```
//Optional chaining with array
// const user = [{ name: 'Lokesh', email: 'lokesh@gmail.
com' }];
const user = [];
console.log(user[0]?.name ?? 'User arry is Empty');
```

#### 4. Sets:

- Set is a collection of unique values, meaning no duplicate are allowed.
- Set provide efficient ways to store and manage distinct elements.
- Set is storing all type of iterable.
- Set support operation like adding, deleting and checking presence of items, enhancing performance for tasks requiring uniqueness.
- **Syntax:**

**const variableName = new Set([iterables]);**

Eg:

```
const orderSet = new Set([
  'Pizza',
  'Pasta',
  'Pizza',
  'Risotto',
  'pizza',
  'Pasta',
]);
console.log(orderSet); //Set(4) { 'Pizza', 'Pasta',
'Risotto', 'pizza' }
```

- If the parameter is not specified or null is passed, then a new set created is empty set.

- **.has()** method: Checking value/element present in Set or not

```
//checking element present in the Set or not
//.has() -->true/false
console.log(orderSet.has('Pizza')); //true
console.log(orderSet.has('Bread')); //false
```

**Note:** If we want to work with unique value and order of element is not important then go with **Set** but if duplicates are allowed and order of element is important than go with **Array**.

## 5. Maps:

- Collection of elements in key and values pair
- It is not similar to **objects** because there is certain difference available in **objects** and **maps**.
- In map **keys** can have any type but in object **keys** are only **string** type.
- On iterating a map object returns the key, and value pair in the same order as inserted.
- Map() constructor is used to create a map.
- Syntax:

```
const varName = new Map([key, value]);
```

Eg:

```
// Creating a Map for product prices
const prices = new Map([
  ["Laptop", 1000],
  ["Smartphone", 800],
  ["Tablet", 400]
]);
```

- **Map.set():** set() method is use to add element in map.

```
const rest = new Map();
rest.set('name', 'Classico Italiano'); //Key with string dat
rest.set(1, 'Firenze,Italy'); //Key with integer data types
```

**Map.set()** method also return a map

```
//Set method also return map  
console.log(rest.set(2, 'Lisbon,Portugal'));
```

```
Map(8) {  
  'name' => 'Classico Italiano',  
  1 => 'Firenze,Italy',  
  2 => 'Lisbon,Portugal',  
  'categories' => [ 'Italian', 'Pizzaria', 'Vegetarian', 'Organic' ],  
  'open' => 11,  
  'close' => 23,  
  true => 'we are open :)',  
  false => 'We are close'  
}
```

We can also use chaining with set method to add multiple elements in Map.

```
rest  
  .set('categories', ['Italian', 'Pizzaria', 'Vegetarian',  
  .set('open', 11)  
  .set('close', 23)  
  .set(true, 'we are open :)')  
  .set(false, 'We are close');  
console.log(rest);|
```

- **Convert object to map**

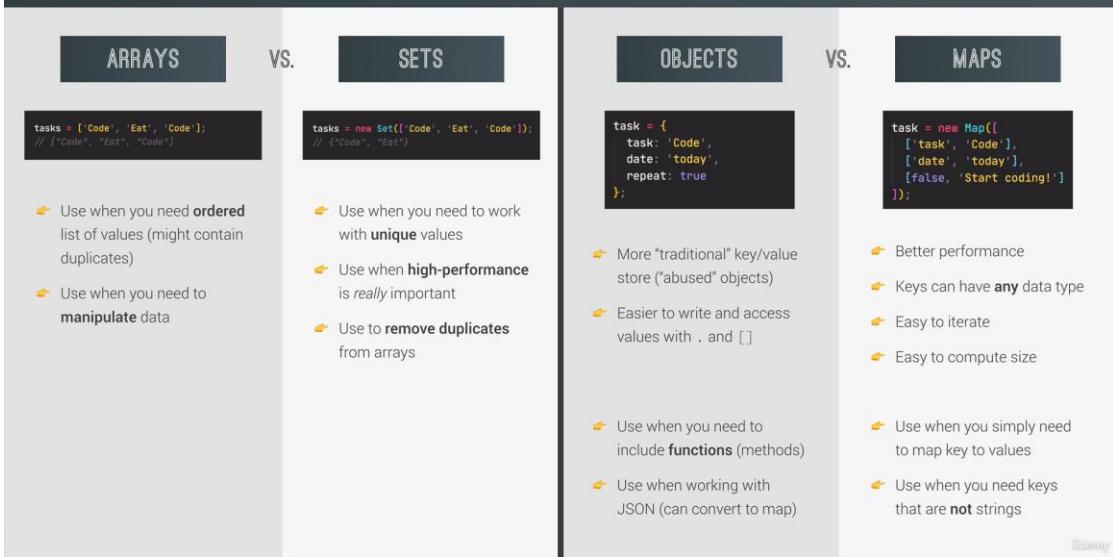
```
//Convert object to map  
const data = { name: 'Lokesh', age: '23' };  
console.log(Object.entries(data)); // [ [ 'name',  
  'Lokesh' ], [ 'age', '23' ] ]  
const mapData = new Map(Object.entries(data));  
console.log(mapData); // Map(2) { 'name' => 'Lokesh', 'age'  
=> '23' }|
```

- **Convert map to array:**

```
//Convert map to array  
const mapToArray = [...question]  
console.log(mapToArray);
```

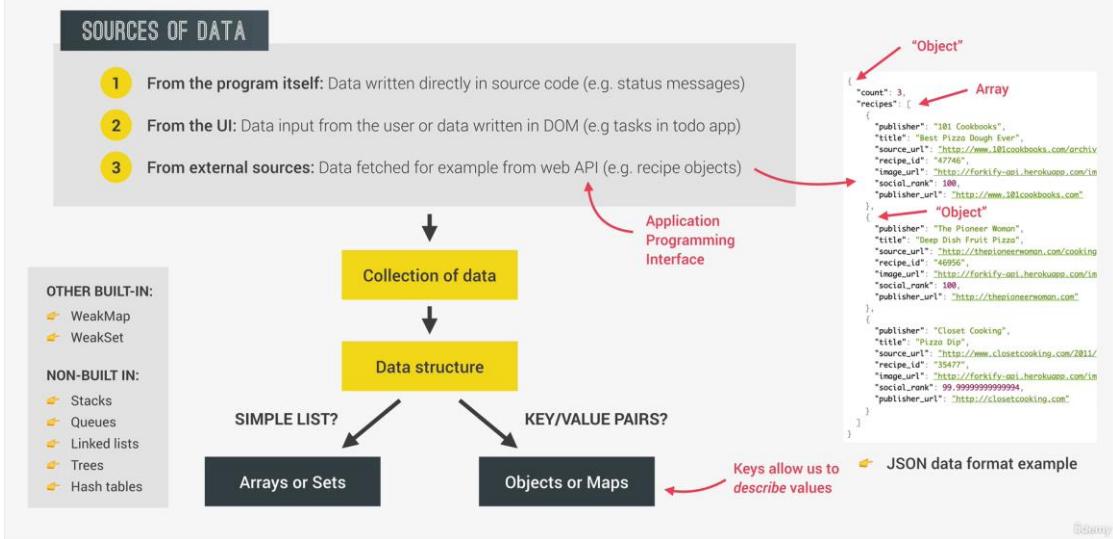
- **When to use which data structure:**
  - In JavaScript there are 4 Data Structures available array, object, set, map
- 1. **Array:**
  - If we want data in simple list and data may be duplicate and order of element is important then use array.
  - Use when you need to manipulate data because there is lots of useful method in array.
- 2. **Set:**
  - If we want data in simple list but data is unique, and order of element is not important than use set.
  - Use when **high-performance** is important.
  - Use to remove duplicates from arrays.
- 3. **Object:**
  - More “traditional” key/value store
  - Easier to write and access value with **. and []**
  - Use when you need to include **functions** (methods) in object
  - Use when working with **JSON** (can convert to map)
- 4. **Map:**
  - It gives Better perform than object
  - Key can have **any** data type
  - Easy to iterate because Map comes from iterable family.
  - Easy to compute size.
  - Use when you simply need to map key to values
  - Use when you need keys that are **not** strings.

## ARRAYS VS. SETS AND OBJECTS VS. MAPS



udemy

## DATA STRUCTURES OVERVIEW



udemy

### ➤ String:

- Collection array of character.
- To find length of string use **string.length** property.

```
//To find length of string
console.log(airline.length); //16
console.log('B339'.length); //4
```

- To find first found index of character or word use **.indexOf()** method.

```
//indexOf() method -->find first found index of letter
console.log(airline.indexOf('r')); //6
console.log(airline.indexOf('Air'))); //4
```

- To find last found index of character/word use **.lastIndexOf()** method.

```
//lastIndexOf() method --> find last found index of letter
console.log(airline.lastIndexOf('r'))); //10
```

- **.slice():** It is use to extract string from original string

```
//slice() method --> it extract string from given index
//It always return a new string which is substring of
original string and slice() is not modify original string
console.log(airline.slice(4)); //Air Portugal

console.log(airline.slice(4, 7)); //Air--> start and end
index

//Extract first and last word dynamically from string
console.log(airline.slice(0, airline.indexOf(' '))); //TAP
console.log(airline.slice(airline.lastIndexOf(' ') +
1)); //Portugal

//use slice with negative value
console.log(airline.slice(-2)); //al
console.log(airline.slice(1, -1)); //AP Air Portuga-->
print remaining letter after removing 1 letter
```

- **.trim() method:** Remove start and end spaces from string

```
//trim() methods -->Remove spaces from start and end of
string
const str = ' Lokesh Jangale ';
console.log(str.trim());
console.log(str.trimStart()); //Remove spaces from start
console.log(str.trimEnd()); //Remove spaces form end
```

- **Split() method:** It divide a string into multiple part from given character and store in array

```
//Split() --> Divide a string in multiple part from given
character and store in array
console.log('a+very+nice+string'.split('+')); //['a',
'very', 'nice', 'string' ]
console.log('Lokesh Jangale'.split(' ')); //['Lokesh',
'Jangale' ]
const [firstName, lastName] = 'Lokesh Jangale'.split(' ');
console.log(firstName, lastName);
```

- **Join() method:** Join splited string in single string

Compress multiple string in single string

```
//Join() --> Compress multiple string in single string
const newName = ['Mr.', firstName, lastName.toUpperCase()].join(' ');
console.log(newName); //Mr. Lokesh JANGALE
```

## ➤ Default Parameter:

```
const bookings = [];
const createBooking = function (
  flightNum,
  numPassenger = 1, //Default value if not any value/undefined is pass
  price = 1000 * numPassenger //Perform calculation as default value
) {
  const booking = {
    flightNum,
    numPassenger,
    price,
  };
  console.log(booking);
  bookings.push(booking);
};

createBooking('LH123'); //{ flightNum: 'LH123', numPassenger: 1, price: 1000 }
createBooking('LH123', 2); //{ flightNum: 'LH123', numPassenger: 2, price: 2000 }
createBooking('LH123', 3, 1500); //{ flightNum: 'LH123', numPassenger: 3, price: 1500 }
//skip parameter to get default value
createBooking('LH123', undefined, 3000); //{ flightNum: 'LH123', numPassenger: 1, price: 3000 }
```

## ➤ Primitive vs Reference type in function parameter:

### ✚ Note: JavaScript is pass by value

- When we pass a primitive type variable in function it will create copy of that variable but when we pass reference type variable like objects it will pass reference of that object because reference type is stored in heap and in call stack reference type variable access by there heap memory reference.

```
const flight = 'LH204';
const lokesh = {
  name: 'Lokesh Jangale',
  passport: 322456483683,
};

const checkIn = function (flightNum, passenger) {
  flightNum = 'LH323';
  passenger.name = 'Mr. ' + passenger.name;
  if (passenger.passport === 322456483683) {
    console.log('Checked in');
  } else {
    console.log('Wrong Passport!');
  }
  console.log(passenger);
};
checkIn(flight, lokesh); // { name: 'Mr. Lokesh Jangale', passport: 322456483683 }
// checkIn(flight, { ...lokesh }); // { name: 'Lokesh Jangale', passport: 322456483683 }
console.log(flight); // LH204
console.log(lokesh); // { name: 'Mr. Lokesh Jangale', passport: 322456483683 } --> It is reference type so the
reference is pass to the function and function can manipulate reference type to avoid this we can pass shallow copy
of object --> checkIn(flight, { ...lokesh });
```

- To avoid this, we pass shallow copy of object by using spread operator

### Syntax:

```
funcName({...objectName});
```

```
checkIn(flight, { ...lokesh }); // { name: 'Lokesh Jangale', passport: 322456483683 }
```

It will pass copy of object not reference of object so if anyone try to make changes then it will not reflect in original object.

```
//This function try to change passport number
//original object: { name: 'Mr. Lokesh Jangale', passport:
322456483683 }
const changePassport = function (passenger) {
  passenger.passport = Math.trunc(Math.random() *
  100000000);
}
// changePassport(lokesh); // { name: 'Mr. Lokesh Jangale',
passport: 86093387 }

changePassport({ ...lokesh });
console.log(lokesh); // { name: 'Mr. Lokesh Jangale',
passport: 322456483683 }
```

➤ **First-class function vs Higher order function:**

## FIRST-CLASS VS. HIGHER-ORDER FUNCTIONS

### FIRST-CLASS FUNCTIONS

- 👉 JavaScript treats functions as **first-class citizens**
- 👉 This means that functions are **simply values**
- 👉 Functions are just another “**type**” of object
  
- 👉 Store functions in variables or properties:

```
const add = (a, b) => a + b;
const counter = {
  value: 23,
  inc: function() { this.value++ },
}
```

  
- 👉 Pass functions as arguments to OTHER functions:

```
const greet = () => console.log('Hey Jonas');
btnClose.addEventListener('click', greet)
```

  
- 👉 Return functions FROM functions
- 👉 Call methods on functions:

```
counter.inc.bind(someOtherObject);
```

### HIGHER-ORDER FUNCTIONS

- 👉 A function that **receives** another function as an argument, that **returns** a new function, or **both**
- 👉 This is only possible because of first-class functions

1 Function that receives another function

```
const greet = () => console.log('Hey Jonas');
btnClose.addEventListener('click', greet)
```

Higher-order function      Callback function      ✅ ☎️ ⚡

2 Function that returns new function

```
function count() {
  let counter = 0;
  return function() {
    counter++;
  };
}
```

Higher-order function      Returned function

- We can say that first – class function is a simply value or a function which is declare in object.
- We can also First – class function as an arguments to another function or return first – class function as a return type.
- However, Higer – order function is a function which receive argument of first-class function. We can simply say that it is a parent function
- Also function that return function is called Higer-order function.

## ➤ Callback Function:

- A callback function is a function passed as an argument to another function, which gets invoked after the main function completes its execution.
- You can pass the main function as an argument, and once the main function finishes its task, it calls the callback function to deliver a result.
- After making any function which can take callback function as arguments so we made this function to perform in more generic way. So we can use that function for multiple logic

Eg:

```
function isOdd(number) {
  return number % 2 != 0;
}

function isEven(number) {
  return number % 2 == 0;
}

function filter(numbers, fn) {
  let results = [];
  for (const number of numbers) {
    if (fn(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));
console.log(filter(numbers, isEven));
```

The above filter() function behave like generic function and take a callback function as a arguments. So this function perform in both cases like isOdd or isEven case.

- A callback function can be an anonymous function, which is a function without a name:

```
function filter(numbers, callback) {
  let results = [];
  for (const number of numbers) {
    if (callback(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, function (number) {
  return number % 2 != 0;
});

console.log(oddNumbers);
```

➤ **Function returning another function:**

```
//Function which return another function
const greet = function (greeting) {
    return function (name) {
        console.log(` ${greeting} ${name}`);
    };
};

const greeterHey = greet('Hey,');
console.log(greeterHey); // [Function (anonymous)]
greeterHey('Lokesh'); // Hey, Lokesh

greet('Hello')('Yeshwant');//Hello, Yeshwant
```

➤ **Call() and apply() method:**

- If we want to explicitly define object to the ‘**this**’ keyword we use call() and apply() method.
- Whereas both methods contain there first element is ‘**this**’ keyword which we will pass.
- But second argument of both keywords are different:
  - Call() method takes normal parameter like whatever parameter we use in function.
  - But apply() method compulsory takes second parameter as array.

Eg

```
'use strict';
//Manipulate this keyword of different object
const lufthansa = {
    airline: 'Lufthansa',
    iataCode: 'LH',
    bookings: [],
    /book: function(){}
book(flightNum, name) {
    `${name} booked a seat on ${this.airline} flight ${this.iataCode}${flightNum}`;
    this.bookings.push({ flight: `${this.iataCode}${flightNum}`, name });
},
lufthansa.book(239, 'Lokesh Dangale');
lufthansa.book(834, 'Yashwant Patil');
console.log(lufthansa);
const eurowings = {
    name: 'Eurowings',
    iataCode: 'EW',
    bookings: []
};
const book = lufthansa.book;
// book(23, 'Sarah Williams');//undefined because it is only normal function and for every normal function call this keyword is undefined
book.call(eurowings, 232, 'Sarah Williams');//this keyword points eurowings object
//call() function have first parameter is 'this' object which is used to point which object should point
console.log(eurowings);
book.call(lufthansa, 439, 'Sagrika sinha');
console.log(lufthansa);

//Apply method -> apply() method takes second argument array
book.apply(eurowings, [564, 'Vishal Patil']);
console.log(eurowings);
```

- By using these two methods we can manually set **this** keyword to function.

➤ **bind() function:**

- bind() function does not immediately call the function instead it return new function where **this** keyword is bound so it set to whatever value which we will pass to bind() function.
- bind() function does not call the function instead it bind function with given object and return that function with **this** keyword.

Eg:

```
//bind() function
const bookEW = book.bind(eurowings);
bookEW(343, 'Vikram Patil');
console.log(eurowings);
```

bind() function just return function with **this** keyword as a value and **bookEW** store that function as a function expression and then it is call like normal function expression is call.

➤ **Immediately invoke function expression(IIFE):**

```
// IIFE
(function () {
  console.log('This will never run again');
  const isPrivate = 23;
})();

// console.log(isPrivate);

(() => console.log('This will ALSO never run again'))
();
```

- If we want to create a function which will execute only once than we can do like this
- The main reason for IIFE is to create a scope for variables.
- But this technique/pattern are no more use because we can easily create scope with just curly braces {}.
- Only one advantage of IIFE is we can declare **var** variable which has a function scope we cannot access it outside the function but if we create scope using curly braces which is block scope **var** variable can also accessible outside scope

```
(function () {
  console.log('IIFE code');
  var isPrivate = true;
})();
// console.log(isPrivate); //Reference error

{
  console.log('Block Scope');
  var isPrivate = true;
}
console.log(isPrivate); //true|
```

➤ **Closures:**

- A closure is a function that reference variables in the outer scope from its inner scope. The closure preserves the outer scope inside its inner scope.
- Closure is function that can remember all variables that existing in parent function which will use in child function or that function which return by Higher-order function, that's why we can access the variable even after the function execution is over in child function.
- This variable may be remove from scope chain but it will present in closure memory.

Eg:

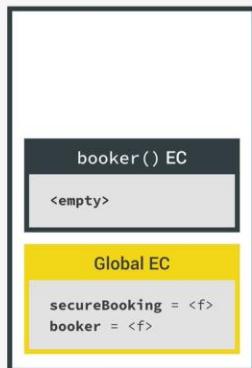
```
const secureBooking = function () {
  let passangerCount = 0;
  let count = 1;
  return function () {
    passangerCount++;
    count++;
    console.log(` ${passangerCount} passengers`);
  };
};

const booker = secureBooking();
booker(); //1 passengers
booker(); //2 passengers
booker(); //3 passengers
console.dir(booker);
```

- In above example booker function can also access variable **count** and **passangerCount** even after execution of **secureBooking()** function is over.
- Because **secureBooking** function return a function which execute by **booker()** and have a access of both variables. Hence js engine directly return the variable environment of **secureBooking** function in **booker** variable and make connection between two execution context, this connection is called as **closure**. That's why booker() function can access variable of secureBooking() function even after execution of secureBooking function is over.

## UNDERSTANDING CLOSURES

- 👉 A function has access to the variable environment (VE) of the execution context in which it was created
- 👉 Closure: VE attached to the function, exactly as it was at the time and place the function was created



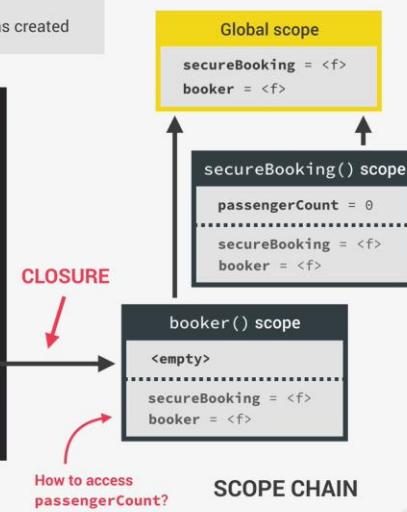
```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`#${passengerCount} passengers`);
  };
};

const booker = passengerCount = 0;

booker(); // 1 passengers
booker(); // 2 passengers
```

This is the function



©demy

## CLOSURES SUMMARY 😊

- 👉 A closure is the closed-over **variable environment** of the execution context in which a function was created, even **after** that execution context is gone;
  - ↓ Less formal
- 👉 A closure gives a function access to all the variables **of its parent function**, even **after** that parent function has returned. The function keeps a **reference** to its outer scope, which **preserves** the scope chain throughout time.
  - ↓ Less formal
- 👉 A closure makes sure that a function doesn't lose connection to **variables that existed at the function's birth place**;
  - ↓ Less formal
- 👉 A closure is like a **backpack** that a function carries around wherever it goes. This backpack has all the **variables that were present in the environment where the function was created**.



- 👉 We do **NOT** have to manually create closures, this is a JavaScript feature that happens automatically. We can't even access closed-over variables explicitly. A closure is **NOT** a tangible JavaScript object.

©demy