JavaScript(Intro,functions,closures,loops,operators,conditional statements and strings)

JavaScript is a high-level, interpreted programming language primarily used to create interactive effects within web browsers. It was developed to enable dynamic content on web pages, enhancing the user experience. It is an essential part of web development alongside HTML (for structure) and CSS (for styling).

1) Press Shift + Alt + F (Windows/Linux) or Shift + Option + F (Mac) to format the code automatically.

JavaScript is **High-level , Garbage-collected , Interpreted or Just-in-time compiled , Multi-paradigm ,Prototype-based Object-oriented ,First-class functions , Dynamic ,Single threaded and Non-blocking event loop.**

1. High-Level

- **Definition**: High-level languages are user-friendly, abstracting away many low-level details like memory management and hardware specifics. JavaScript allows developers to focus on application logic rather than system details.

let message = "Hello, World!";// You don't need to manually manage memory for the variable.

console.log(message);

Languages like C or C++ are low-level and require manual memory management (using malloc() or free()), while JavaScript automatically handles memory allocation and garbage collection

2. Garbage-collected

JavaScript has an automatic garbage collector that takes care of memory management by reclaiming unused memory from objects no longer in use, reducing the chance of memory leaks.

let obj = { name: "John" }; // obj references memory

obj = null; // Memory for obj can be reclaimed by the garbage collector

In languages like C or C++, developers must manually manage memory allocation and deallocation using functions like `malloc` and `free`.

3. **Interpreted or Just-in-time compiled (JIT)**

JavaScript is typically interpreted, meaning the code is executed directly by the interpreter. Modern JavaScript engines (like V8) use Just-In-Time (JIT) compilation, where code is compiled into machine code during execution for faster performance.

Example in JavaScript:

- Interpreted: A JavaScript engine (like V8) reads and executes JavaScript code directly.
- JIT Compilation: In modern browsers, JavaScript code is compiled to machine code at runtime.

4. Multi-paradigm

- Explanation: JavaScript supports different programming paradigms like procedural, object-oriented, and functional programming.

  Procedural:

```
function greet(name) {

  console.log("Hello, " + name);

}

greet("John");
```

  Object-Oriented:

```
function Person(name) {

  this.name = name;

}

let person = new Person("John");
```

  Functional:

```
const greet = (name) => console.log(`Hello, ${name}`);

greet("John");
```

  In comparison, Python is also multi-paradigm, supporting both object-oriented and functional programming, whereas languages like Java are predominantly object-oriented

5. Prototype-based Object-oriented

JavaScript is prototype-based, meaning objects can directly inherit from other objects, as opposed to the classical class-based inheritance used in many other languages like Java or C++.

```
const person = {

  name: "John",

  greet() {

    console.log(`Hello, my name is ${this.name}`);

  }

};



const employee = Object.create(person);

employee.name = "Jane";

employee.greet(); // "Hello, my name is Jane"
```

6. First-class Functions

Functions in JavaScript are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.

```
function greet(name) {

  return `Hello, ${name}`;

}

let greetFunction = greet; // Assign function to variable

console.log(greetFunction("Alice"));
```

7. Dynamic

JavaScript is dynamically typed, meaning variable types are determined at runtime, and types can change during execution.

```
let a = 42; // 'a' is a number

a = "Hello"; // 'a' is now a string

console.log(a); // Output: Hello
```

In **Java**, you must declare a variable's type explicitly (e.g., `int x = 10`), and the type cannot change during execution.

8. Single-threaded

JavaScript is single-threaded, meaning it runs in one thread of execution at a time. However, it uses an event loop to handle asynchronous operations.

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

setTimeout(() => console.log("Middle"), 1000);

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

Output

Start

End

Middle

9. Non-blocking Event Loop

JavaScript uses an event-driven, non-blocking I/O model. The event loop allows it to handle asynchronous operations, such as HTTP requests or file reading, without blocking the main thread.

```
async function fetchData() {

  console.log("Fetching data...");

  let data = await fetch('https://api.example.com');

  console.log("Data fetched!");

}
```

```
fetchData();

console.log("This will log before 'Data fetched!' due to async nature.");
```

Javascript Engine

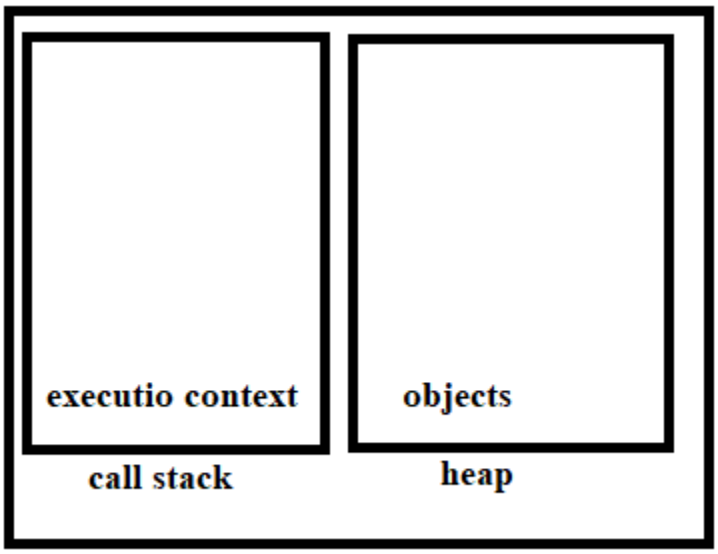
A JavaScript engine is a computer program that executes JavaScript code and converts it into computer understandable language(0s and 1s).

Every browser has its own Javascript Engine. The most well known engine is Google Chrome V8 Engine.

List of JavaScript Engines:

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

Every javascript engine has a **call stack** and **heap.**



Call Stack is used to **execute the source code** using **execution context**.

Heap is where objects are stored (object in memory).

1) **Compilation**

   Entire source code is converted into machine code at once , and written to a binary file that can be executed by a computer.

   Source code —--step1(**compilation**)---------> **portable file(machine code)**-------step2(**execution**)--------->program running.

   Execution can after compilation.

2) **Interpretation:**

   The interpreter runs through the source code and executes it line by line.

   Source code —--step1(**execute line by line**)---------> **program running.**

   Here after execution, code needs to be converted to machine code.

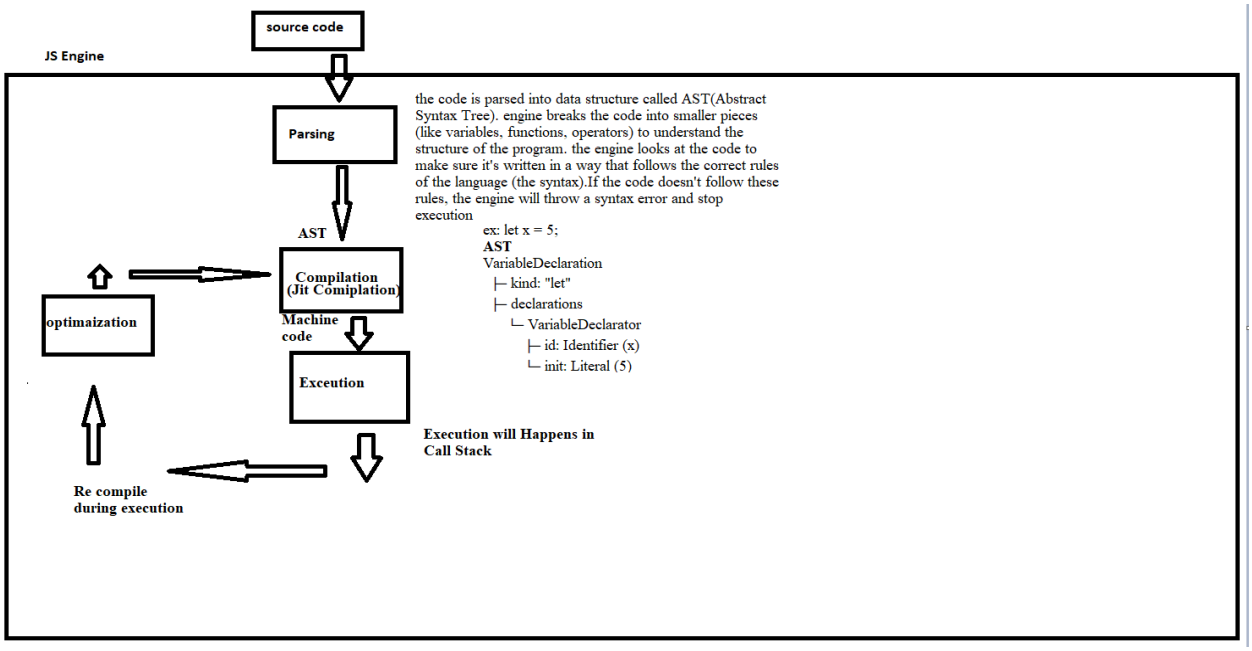
3) **JIT(Just-in-time) compilation:(Javascript used)**

   Entire code is converted into machine code at once , then executed immeditaly. No portable file required .

   Source code —--step1(**compilation**)--------->(**Machine code**)-------step2(**execution**)--------->program running.

   JavaScript is mainly interpreted, but modern JavaScript engines, like V8 in Google Chrome, **use JIT (Just-In-Time) compilation to boost performance**. They convert JavaScript code into optimized machine code right before it runs. This mix of interpretation and JIT compilation makes JavaScript fast and versatile for web applications.

   Modern JIT(Just-in-time) compilation of javascript:



Inside Execution Context:

1)Variable Environment

Let, const and var declarations

Functions

Argument object

2) scope chain

Global scope: outside of function or block, accessible everywhere

```
var globalVar = "I am global"; // Declared outside any function

function testGlobal() {

  console.log(globalVar); // Can access the global variable

}
```

```
testGlobal(); // Output: I am global

console.log(globalVar); // Output: I am global
```

Function scope: accessible only inside function , not outside

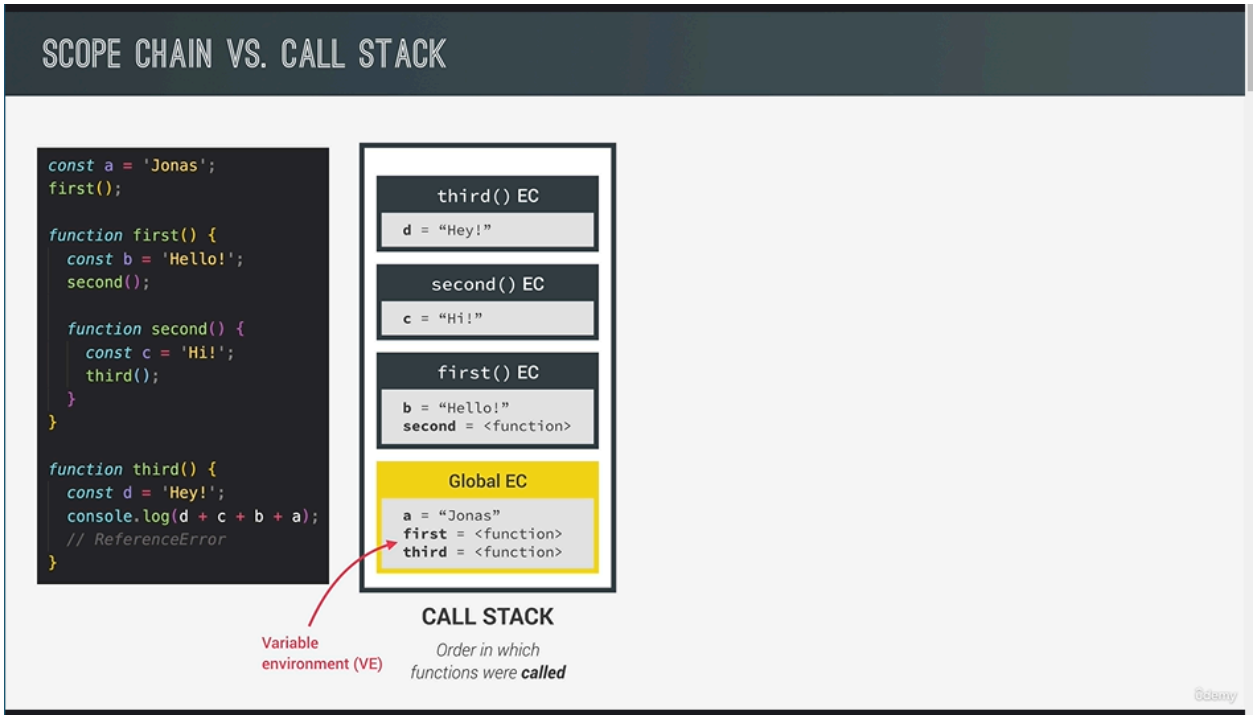
```
function testFunctionScope() {
  var functionVar = "I am inside a function"; // Only accessible within this function
  console.log(functionVar); // Output: I am inside a function
}

testFunctionScope();

console.log(functionVar); // ReferenceError: functionVar is not defined
```

Block scope: inside a block, applies only for let and const variables.function scope also block scoped(only in strict mode), Block-scoped variables are defined using `let` or `const` and are only accessible within the block (enclosed by `{}`) in which they are declared.

```
{
  let blockVar = "I am inside a block";
  const blockConst = "I am also inside a block";
  console.log(blockVar); // Output: I am inside a block
}

console.log(blockVar); // ReferenceError: blockVar is not defined
```

Global Scope > **Function Scope** > **Block Scope**.

- Global scope variables are accessible everywhere.
- Function scope variables are accessible only inside the function.
- Block scope variables (declared with `let` or `const`) are accessible only inside the block.



Hoisting in JavaScript

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope (either function or global) during the compilation phase, before the code has been executed. However, only the declarations (not the initializations) are hoisted.

How hoisting works for variables:

1. **var declaration:** When a variable is declared using var, its declaration is hoisted to the top, but its value is not assigned until the code execution reaches that point.

   Example:
   ```
   console.log(x);  // undefined, not ReferenceError

   var x = 5;

   console.log(x);  // 5
   ```

   ```
   undefined                    script.js:41
   5                            script.js:43
   ```

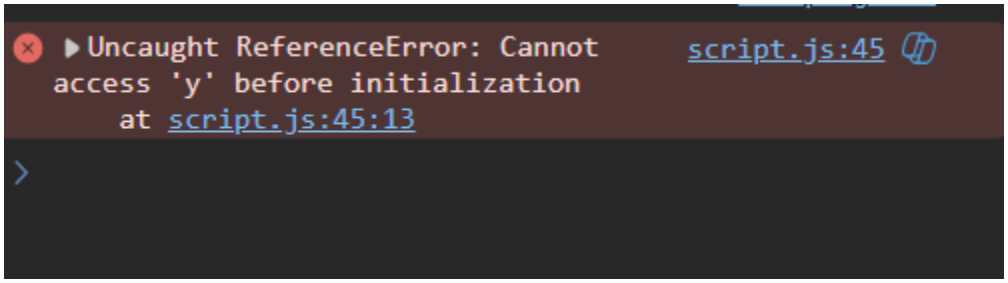
   Explanation:

- The declaration var x is hoisted to the top.
- However, the initialization (x = 5) occurs where it is written, so x is undefined when logged before the initialization.

2. **let and const declarations:** let and const declarations are hoisted to the top of their block scope, but they are not initialized until the code execution reaches their actual declaration. This creates the **Temporal Dead Zone (TDZ)**.

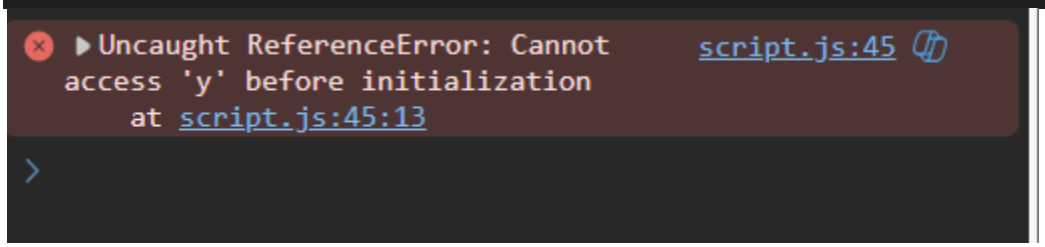
   Example:
   console.log(y); // ReferenceError: Cannot access 'x' before initialization

   let y = 5;



Ex2:

```
console.log(y); // ReferenceError: Cannot access 'x' before initialization

const y = 5;
```



Temporal Dead Zone (TDZ)

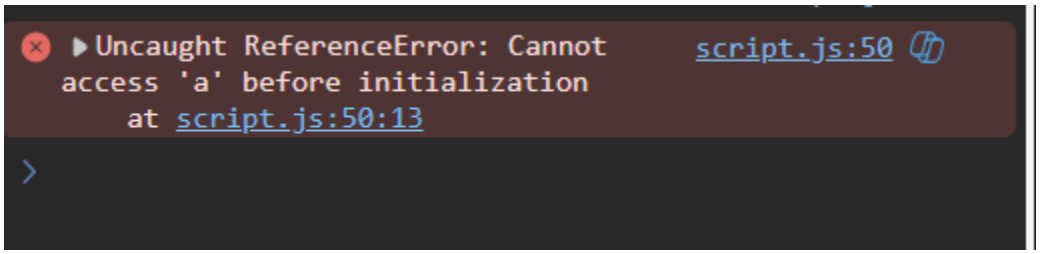
The Temporal Dead Zone is the time between the **entering of the scope** and **the actual initialization of variables declared with let and const**. During this period, any reference to the variable will result in a ReferenceError.

How TDZ works:

- If you try to access a let or const variable before it has been initialized, it results in a ReferenceError.

Example:

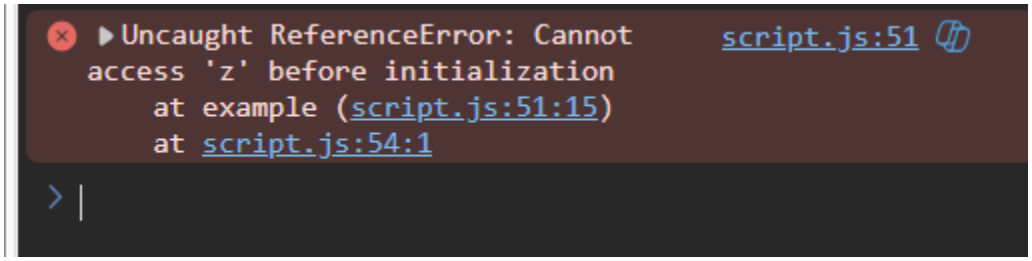
```
console.log(a);  // ReferenceError: Cannot access 'a' before initialization

let a = 10;
```



In the above example, even though a is hoisted, it is in the TDZ until the line let a = 10; is executed.

Example with let and TDZ:

```
function example() {
  console.log(z);  // ReferenceError
  let z = 5;  // 'x' is in TDZ here
}
example();
```



Explanation:

- let x is hoisted but remains uninitialized.
- Trying to access x before it's initialized causes a ReferenceError due to the Temporal Dead Zone.

Function Hoisting:

Function declarations are hoisted entirely, meaning both the declaration and the definition are moved to the top.

Example:

```
myFunction();  // Works fine!

function myFunction() {
  console.log("Hello, World!");
}
```

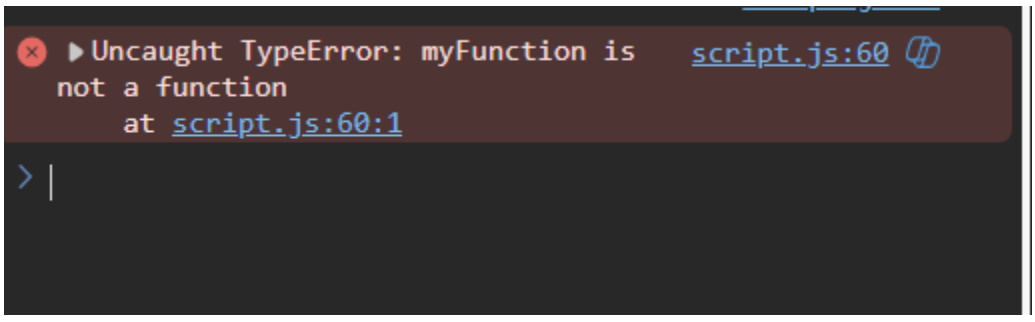
Explanation:

- The entire function declaration is hoisted, so the function can be called before its definition.

However, **function expressions** (e.g., assigning a function to a variable) are not hoisted.

Example:

```
myFunction();  // TypeError: myFunction is not a function

var myFunction = function() {
  console.log("Hello, World!");
};
```



Explanation:

- Only the variable myFunction is hoisted, but not the function definition, which results in a TypeError when trying to call it before the assignment.

Summary:

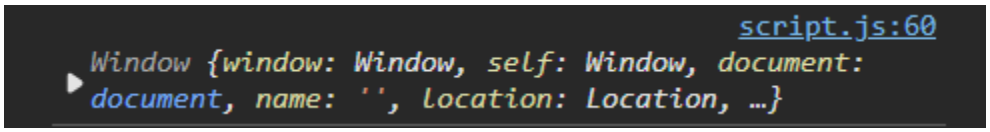
- **Hoisting** applies to variable and function declarations, but var variables are initialized as undefined while let and const are not initialized until their definition.
- The **Temporal Dead Zone (TDZ)** occurs with let and const, where variables cannot be accessed before their initialization.
- Function declarations are fully hoisted, but function expressions (assigned to variables) are not.

This Keyword

This keyword refers to the object it belongs to. Its value depends on how and where it is invoked.

1)In Global Context

```
console.log(this); // Window object
```



2) Inside a Function

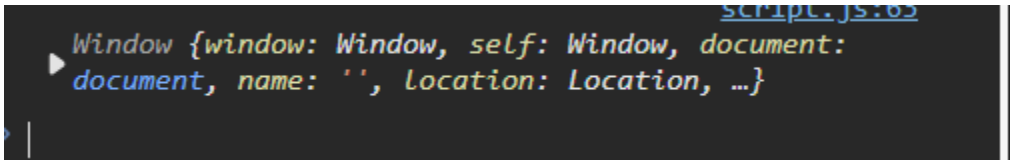
a) **Strict mode:**

```
function showThis() {
   console.log(this);
}
showThis(); // Window (in browsers) or global object (in Node.js)
```

b) **Non-strict mode**

```
function showThis() {
```

```
    console.log(this);
}
showThis(); // Window (in browsers) or global object (in Node.js)
```



3) Inside an Object Method

this refers to the object the method belongs to:

```
const obj = {
  name: 'John',
  greet() {
    console.log(this.name);
  },
};
obj.greet(); // "John"
```

Ex2

```
const obj = {
  name: 'John',
  greet: function () {
```

```
    console.log(this);

  },

};

obj.greet();
```

Output

4)**Arrow Functions**

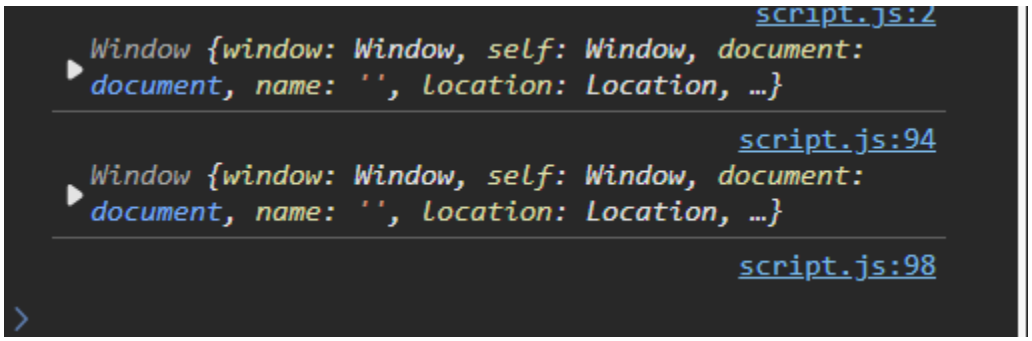
Arrow functions do not have their own `this`. Instead, they inherit it from the surrounding lexical scope:

const obj = {

  name: "John",

  greet: () => {

```
    console.log(this.name);
  }
};
obj.greet(); // undefined (in browsers) because `this` refers to the global object.
```



As it using inherit it from the surrounding lexical scope:

parent value of this keyword (window)

5) In a Constructor Function

`this` refers to the new object being created:

```
function Person(name) {
  this.name = name;
}
const john = new Person("John");
console.log(john.name); // "John"
```

6) In Classes

`this` behaves similarly to a constructor:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    greet() {
        console.log(this.name);
    }
}
const john = new Person("John");
john.greet(); // "John"
```

8. Event Listeners

In regular functions, `this` refers to the element that received the event:

```
document.querySelector("button").addEventListener("click", function() {
    console.log(this); // The button element
});
```

arguments keyword

The arguments keyword in JavaScript is used to represent an array-like object that holds all the arguments passed to a function. Its behavior differs between **regular functions** and **arrow functions**.

```
function add(a, b) {
```

```
  console.log(arguments);

  const c = a + b;

  console.log(c);

  return c;

}

add(2, 3);

add(8, 3, 5, 6, 8);
```

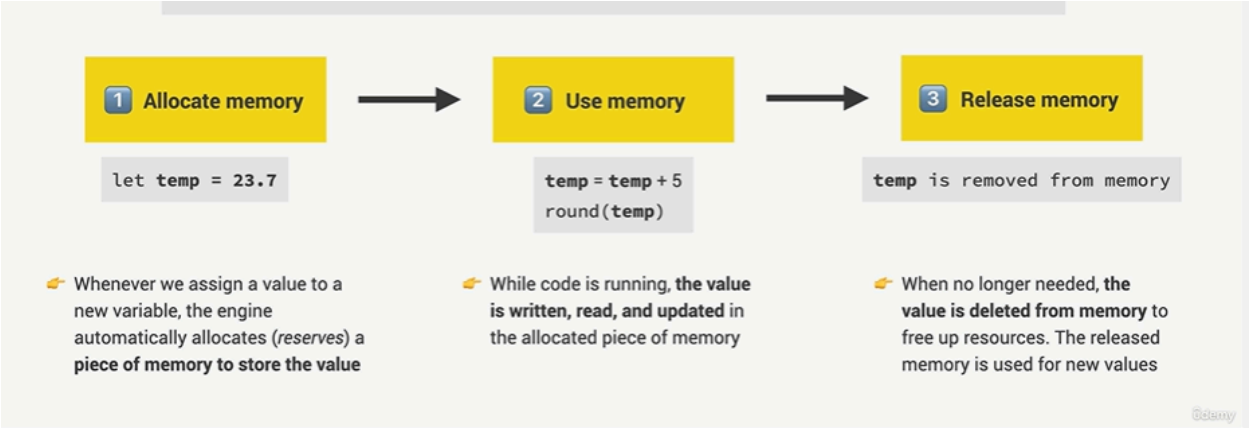
```
                                                    script.js:104
▼ Arguments(2) [2, 3, callee: (...), Symbol(Symbol.ite
  rator): ƒ] i
    0: 2
    1: 3
    callee: (...)
    length: 2
  ▶ Symbol(Symbol.iterator): ƒ values()
  ▶ get callee: ƒ ()
  ▶ set callee: ƒ ()
  ▶ [[Prototype]]: Object
5                                                   script.js:106
                                                    script.js:104
▼ Arguments(5) [8, 3, 5, 6, 8, callee: (...), Symbol(S
  ymbol.iterator): ƒ] i
    0: 8
    1: 3
    2: 5
    3: 6
    4: 8
    callee: (...)
    length: 5
  ▶ Symbol(Symbol.iterator): ƒ values()
  ▶ get callee: ƒ ()
  ▶ set callee: ƒ ()
  ▶ [[Prototype]]: Object
11                                                  script.js:106
```

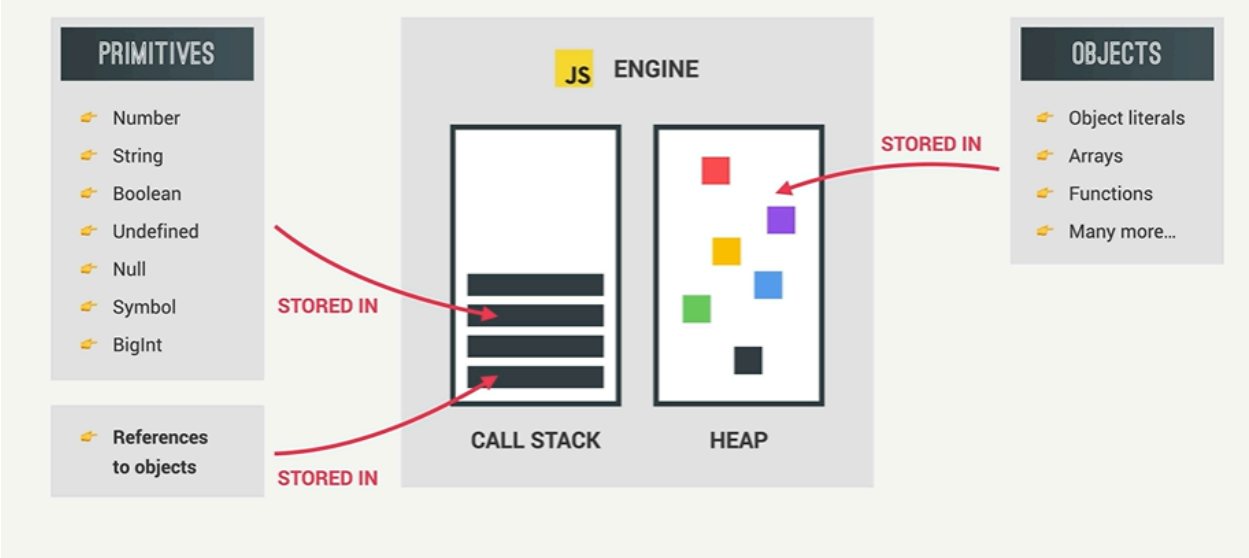
Arrow function

```
//in arrow function



const summ = (a, b) => {

  console.log(arguments);

  const d = a + b;

  console.log(d);

  return d;

};

summ(3, 4);

summ(6, 7, 8, 0);
```

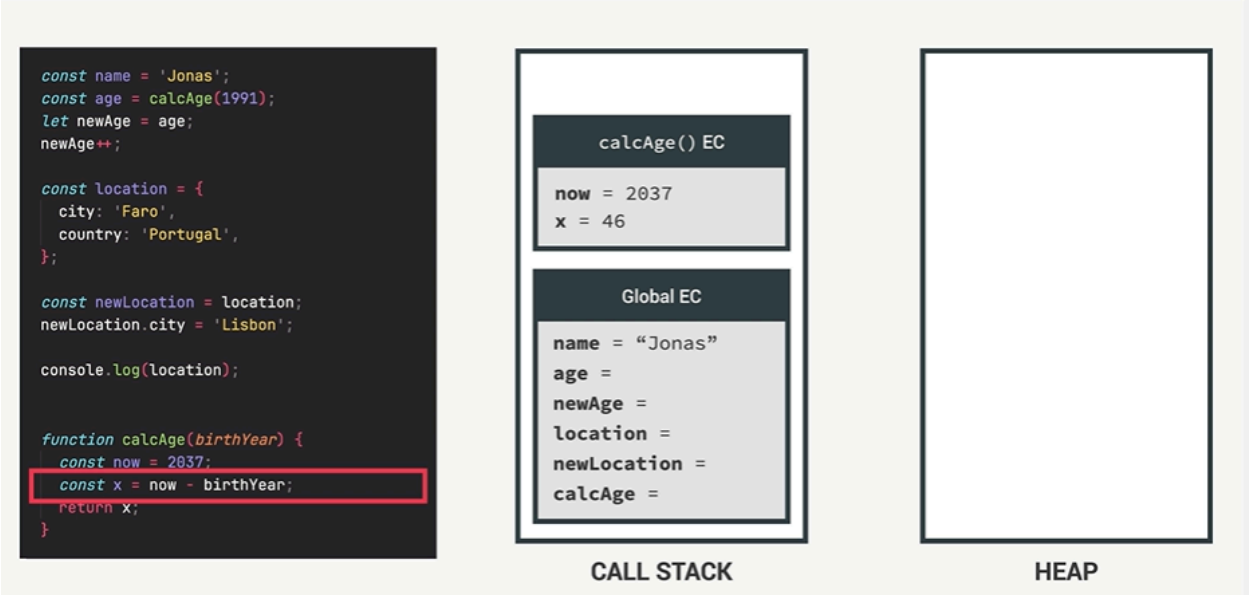
```
▶Uncaught ReferenceError: arguments script.js:115
is not defined
    at summ (script.js:115:15)
    at script.js:120:1
>
```

Memory Management in javascript automatically





Ex1

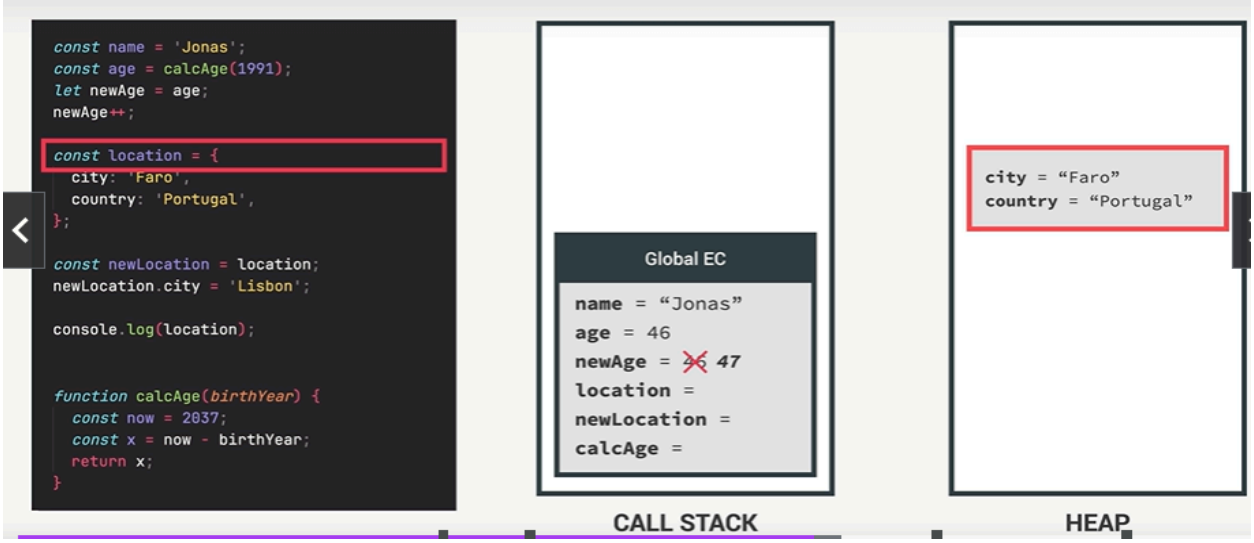


Once the calAge() is executed then it will be from callStack and data will be stored in the respective reference value.(ex age =46)

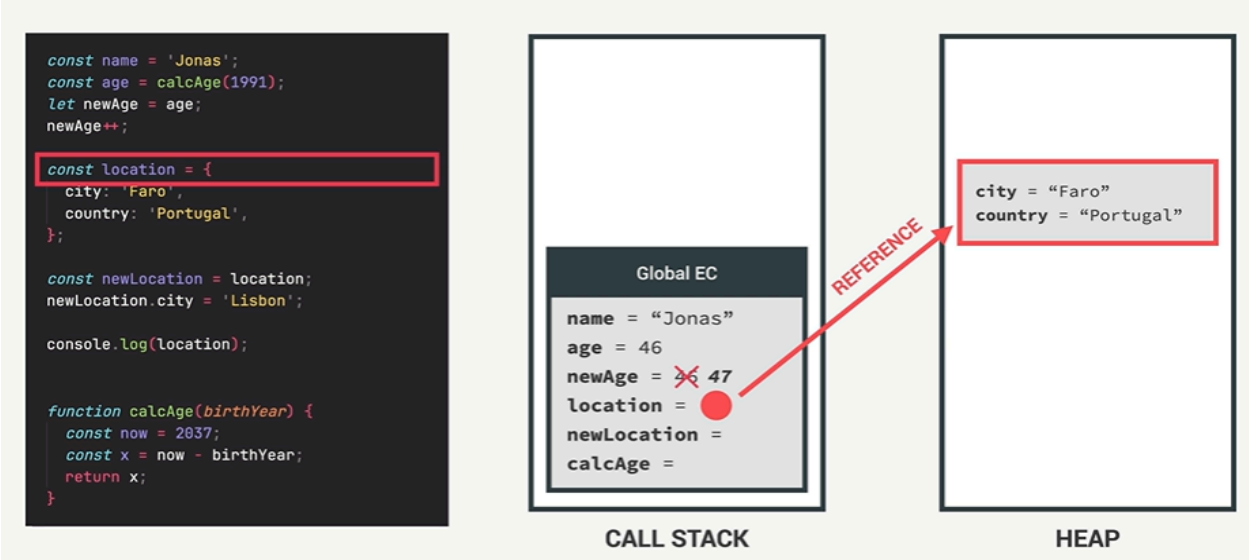
Then newAge =age;

newAge++

Now the location object will be stored in the heap memory area.



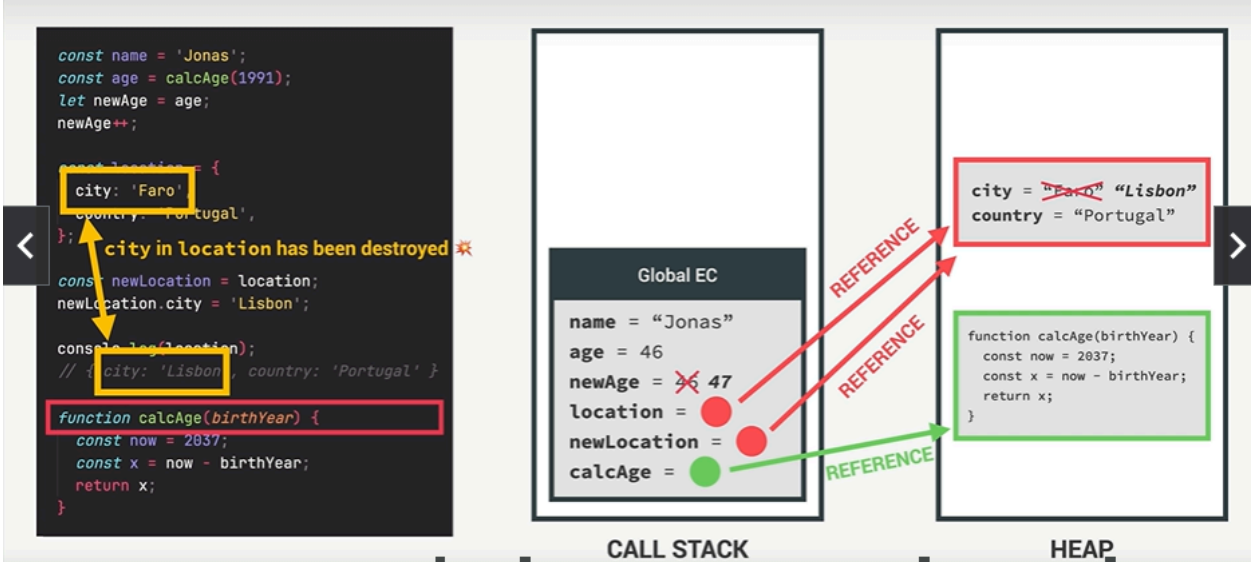
The reference will be stored in the call stack only but the object data will be stored in the heap area.



Const newLocation = location
It means location and newLocation pointing to the same object elements.

newLocation.city='Lisbon'
Here we are updating the city value with new values.

Even calAge() also stored in heap only



Shallow Copy vs Deep copy

What is a Shallow Copy?

A **shallow copy** creates a new object, but it only copies the immediate properties. If a property is a reference to another object (e.g., arrays or nested objects), the shallow copy will still reference the same object in memory.

A shallow copy occurs when you copy the reference of an object to a new variable. In this process, only the top-level properties are copied, while nested objects or arrays still reference the original memory location. This means that if you change the nested properties in one object, those changes will reflect in the other because they share the same memory reference.

Using the spread operator **({ ...obj }) or Object.assign** creates a **shallow copy** of the element's object. This means the top-level properties are copied into a new object, but nested objects or arrays remain **referenced** (not duplicated)

1) const newCopy = Object.assign({}, elemnts); // using Object.assign

Or

2) const newCopy = { ...elemnts }; //using spread operator

Ex1:

```
const elemnts = {
  firstName: 'ram',
  lastName: 'j',
  age: 24,
  fav: ['cricket', 'flowers'],
};
//shallow copy
console.log('before', elemnts);
const newCopy = { ...elemnts }; //it will copy entire object
newCopy.lastName = 'jk';
newCopy.fav.push('music');
console.log('after changes original value', elemnts);
console.log('after changes', newCopy);
```

Output

Primitive Properties: For properties like firstName, lastName, or age, a shallow copy duplicates their values because they are primitives. Changing newCopy.lastName does not affect elemnts.lastName.

Nested Objects or Arrays: For properties like fav, which is an array, a shallow copy only copies the reference. Both newCopy.fav and elemnts.fav point to the same array in memory. Any modifications to the array affect both objects.

What is a Deep Copy?

A **deep copy** creates a completely independent copy of the object and all nested structures, ensuring no shared references between the original and the copied object.

A deep copy, on the other hand, creates a completely independent copy of the object, including all nested objects or arrays. This ensures that changes made to one object do not affect the other. Each object is stored in a separate memory location, making them entirely independent.

structuredClone is a built-in JavaScript method that performs a **deep copy** of an object, including its nested objects and arrays.

structuredClone is supported in most modern browsers (since Chrome 98, Firefox 94, and Node.js 17). For older environments, a library like Lodash (_.cloneDeep)
or
JSON.parse(JSON.stringify()) can be used as an alternative.
Now to create a deep copy of an object in JavaScript we use JSON.parse() and JSON.stringify() methods.

```
let newEmployee = JSON.parse(JSON.stringify(employee));
```

```
const elemnts = {
  firstName: 'ram',
  lastName: 'j',
  age: 24,
  fav: ['cricket', 'flowers'],
};
//deep copy
const newClone = structuredClone(elemnts);
newClone.lastName = 'jk';
newClone.fav.push('music');
console.log('before', elemnts);
console.log('after changes', newClone);
```

Output

console.log and document.write

console.log and document.write are both methods used for outputting information in JavaScript, but they serve different purposes and behave differently.

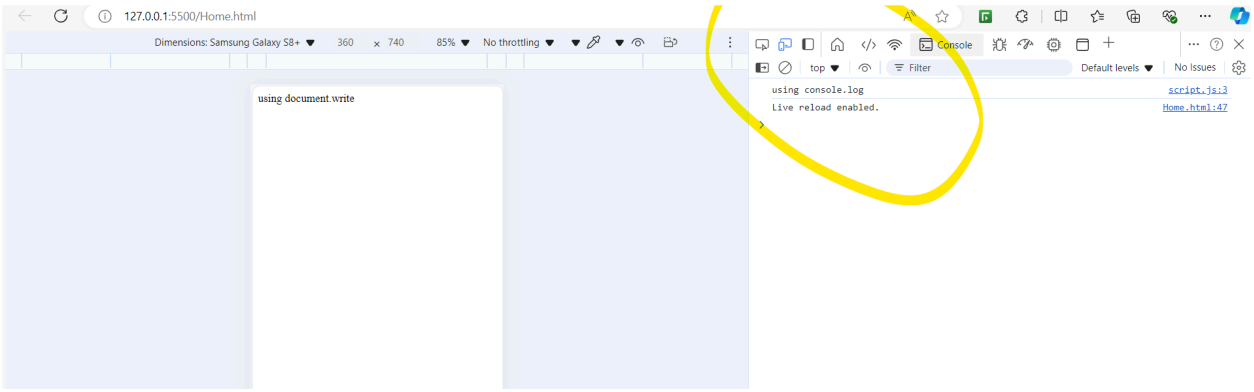
1. console.log()

- **Purpose**: It is used to log messages to the browser's developer console (typically accessible through the browser's DevTools).
- **Use Cases**:
    - Debugging and development purposes.
    - Printing information such as variables, objects, and messages to track the flow of your script.
- **Behavior**: It does not affect the content of the web page itself. It's only visible to developers through the browser console.

```
console.log('Hello, world!');
```

In this example, the message 'Hello, world!' will be displayed in the browser's developer console but will not appear on the page.

```
alert("alert is coming")
console.log("using console.log");

document.write("using document.write")
```



2. document.write()

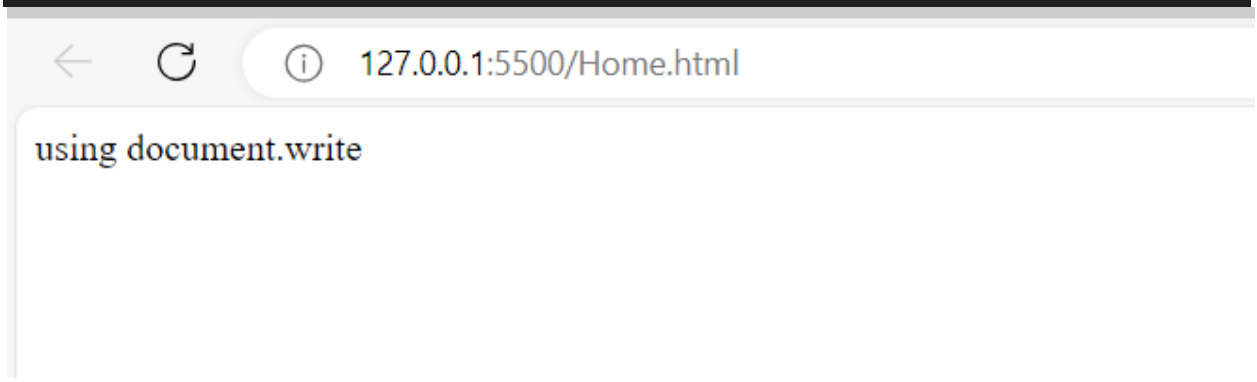
- **Purpose**: It is used to directly write text or HTML content to the document (web page).
- **Use Cases**:
    - Adding content to a webpage during the page load.
    - Can be used to dynamically add HTML to the page (though it is not recommended in modern web development).

- **Behavior**: It can modify the content of the page. If used after the document has fully loaded, it can overwrite the entire page content, which is why it is considered an outdated and risky practice.

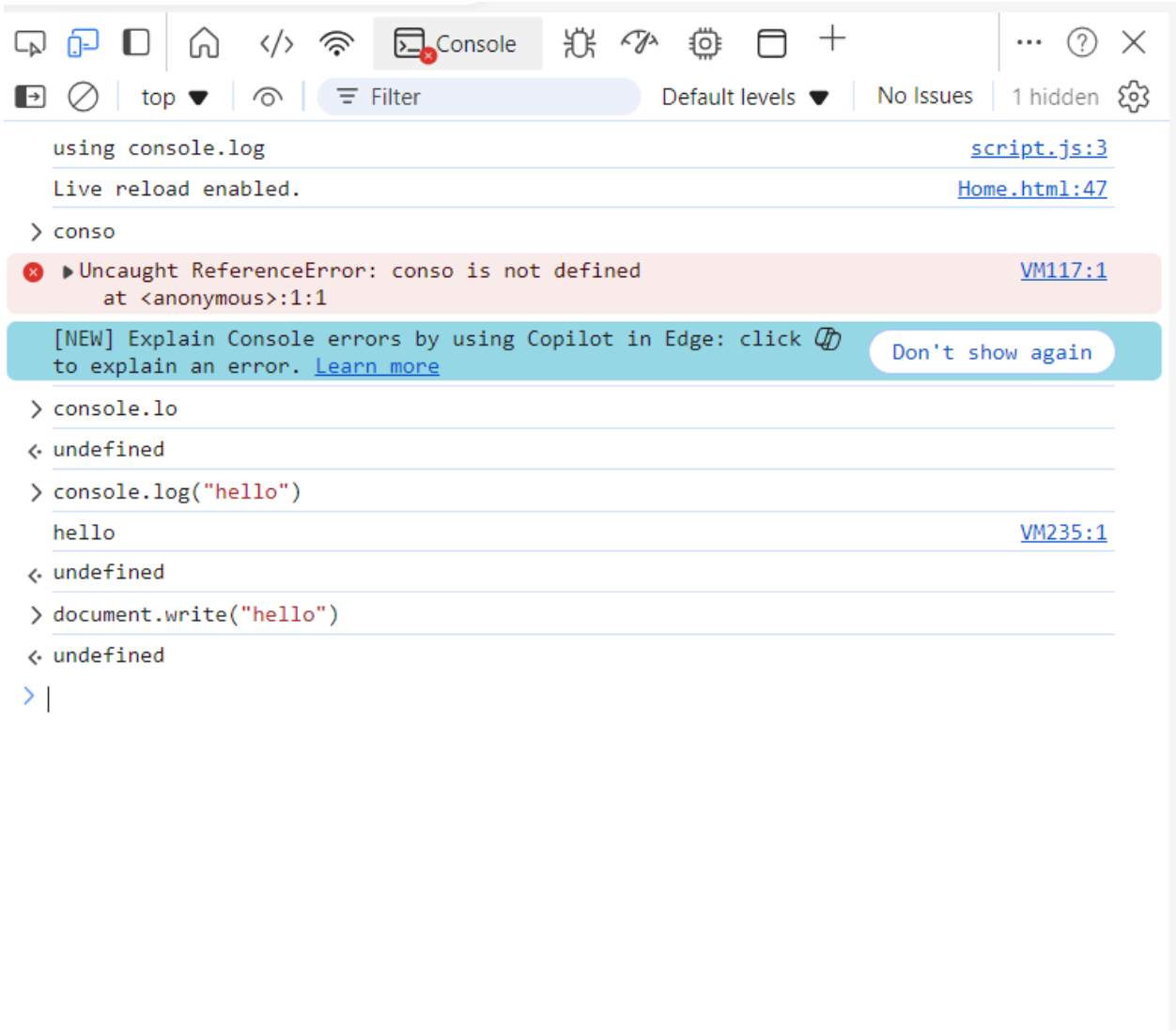
```
document.write('Hello, world!');
```

In this case, the message 'Hello, world!' will appear directly on the web page, replacing any existing content if used after the page has loaded.

```
document.write("using document.write")
```



We can also write console.log ,document.write directly in the console and execute .



Variable

In JavaScript, **variables** are used to store data values. A variable can hold different types of data, such as numbers, strings, objects, arrays, etc.

1) Variable names can start with a letter (a-z, A-Z), an underscore (_), or a dollar sign ($).

They cannot start with a number.

let _myVariable; // valid

let $myVar; // valid

let myVar1; // valid

let 1myVar; // invalid

let my_variable = 10; // valid

let myVar$ = 20; // valid

2) JavaScript has reserved words (like let, const, function, class, var, etc.) that cannot be used as variable names.
3) JavaScript is case-sensitive, so myVar and myvar would be considered two different variables.
4) Variable names cannot contain spaces. If you want to separate words in a variable name, use camelCase (e.g., myVariableName) or underscores (e.g., my_variable_name).

   let myVariable = 15; // valid

   let my variable = 15; // invalid

1. var (Function-scoped or globally scoped)

- **Scope**: var is function-scoped, meaning it is accessible within the function where it is declared (if declared inside a function) or globally if declared outside any function.
- **Redeclaration**: var allows redeclaration within the same scope without throwing an error.
- After the declaration, the variable has no value (technically it is undefined).
- If we print variables before declaration then we will get undefined.

When to use: Rarely, and only if you're working with older codebases or require function-scoped variables for compatibility.

Scope: Function-scoped (not block-scoped).

Issues:

Can lead to bugs due to **hoisting** and lack of block scope.

Allows redeclaration, which can make code harder to debug.

```
// var

console.log(salary); // printing first then declaring
```

```
var salary; // value not assigned

console.log(salary); // will get undefined



var name = 'Alice';  // Declare a variable with 'var'

console.log(name);

if (true) {

    var name = 'Bob';  // Redeclare the same variable (function-scoped)

    console.log(name);  // Outputs: Bob

}



console.log(name);  // Outputs: Bob (redeclaration with 'var' affects the entire scope)
```

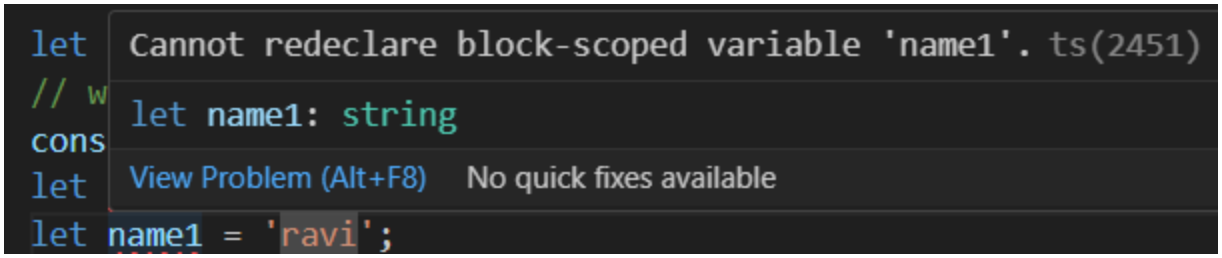
2) let (Block-scoped)

- **Scope**: let is block-scoped, meaning it is only available within the block (a pair of {}) where it is defined, such as within loops or conditionals.
- **Redeclaration**: let cannot be redeclared within the same scope.

  When to use: For variables that need to be reassigned (mutable) but are limited to block scope.

  Scope: Block-scoped (limited to {} blocks).

  Safer Alternative to var: It doesn't allow redeclaration within the same scope.



- After the declaration, the variable has no value (technically it is undefined).
- If we print variables before declaration we will get errors.
- lock-scoped: restricted to the block (loops, if statements, functions) where it is declared.
- Cannot be redeclared in the same scope.

```
let salary1; // value not assigned
// will get undefined
console.log(salary1);
let name1 = 'ravi';  // Declare a variable with 'let'
console.log(name1); //ravi
if (true) {
    let name1 = 'rani';  // This 'let' is block-scoped
    console.log(name1);  // Outputs: rani (block-scoped)
}

console.log(name1);  // Outputs: ravi (does not affect the outer variable)
```

3) const (Block-scoped, Read-Only)

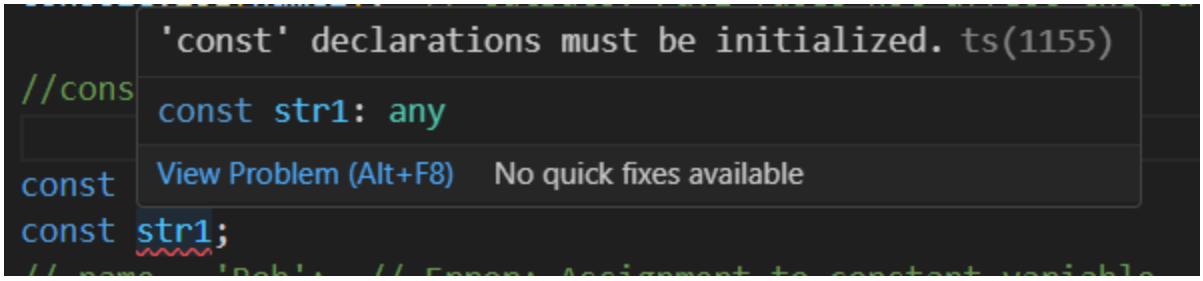
- **Scope**: const is also block-scoped like let.
- **Redeclaration**: const cannot be redeclared within the same scope.
- **Reassignment**: const does not allow reassignment after the initial assignment. It is read-only after being assigned a value.
- For const value should be initialized.

  When to use: For variables whose reference won't change (immutable references).

  Note: This doesn't make the content of objects or arrays immutable.

  Scope: Block-scoped (like let).

  Reassignment: Not allowed after declaration.




```
//const

const str = 'hello';  // Declare a constant

// name = 'Bob';  // Error: Assignment to constant variable.

if (true) {
    const str = 'hii';  // This 'const' is block-scoped
    console.log(str);  // Outputs: Bob
}

console.log(str);  // Outputs: Alice (does not affect the outer variable)
```

4. var can be redeclared in the same scope, but let and const cannot be

JavaScript

```
1  var x = 10;
2  var x = 20; // Allowed
3
4  let y = 30;
5  let y = 40; // SyntaxError
6
7  const z = 50;
8  const z = 60; // SyntaxError
```

Data types

1. Primitive Data Types

These are immutable and represent a single value.

1. **Number:** Represents integer and floating-point numbers.

   Example:
   let num = 42;

   let pi = 3.14;

2. **String :**Represents text, enclosed in single ('), double ("), or backticks (`) for template literals.

   Example:
   let greeting = "Hello, world!";

   let str =’hii’;

   let name = `John`;

3. **Boolean**: Represents true or false.

   Example:

   let isActive = true;

   let isLoggedIn = false;

4. **Undefined**:A variable declared but not assigned a value automatically gets the value undefined.

   Example:
   let x;

   console.log(x); // undefined

5. **Null:** Represents an intentional absence of any value (not the same as undefined).
   let data = null;

6. **BigInt**

2) Non-Primitive (Reference) Data Types

These are mutable and can hold multiple values.

1. **Object:** Collection of key-value pairs.

   ```
   let user = { name: "Alice", age: 25 };
   ```

2. **Array:** A special type of object for ordered lists.

   Example:
   ```
   let fruits = ["apple", "banana", "cherry"];
   ```

3. **Function:** A callable object.

   Example:
   ```
   function greet() {
     console.log("Hello!");
   }
   ```

4. **Date:** For handling dates and times.
   - Example:
     ```
     let today = new Date();
     ```

typeof Operator

```
console.log(typeof 42); // "number"

console.log(typeof "Hello"); // "string"
```

Output

```
42                                                                    script.js:53
3.14                                                                  script.js:55
Hello, world!                                                         script.js:60
true                                                                  script.js:64
undefined                                                             script.js:67
null                                                                  script.js:70
▶ {name: 'Alice', age: 25}                                            script.js:74
▶ (3) ['apple', 'banana', 'cherry']                                   script.js:77
Hello!                                                                script.js:80
Wed Nov 27 2024 15:36:39 GMT+0530 (India Standard Time)               script.js:85
number                                                                script.js:87
string                                                                script.js:88
```

Operators

There are following types of operators in JavaScript.

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Bitwise Operators
4. Logical Operators
5. Assignment Operators
6. Special Operators

1. Arithmetic Operators

Used for mathematical operations.

Operator	Description	Example	Result	Explanation
`+`	Addition	`5 + 3`	`8`	Adds two values.
`-`	Subtraction	`5 - 3`	`2`	Subtracts the second value from the first.
`*`	Multiplication	`5 * 3`	`15`	Multiplies two values.
`/`	Division	`6 / 3`	`2`	Divides the first value by the second.
`%`	Modulus (Remainder)	`5 % 2`	`1`	Returns the remainder of division.
`**`	Exponentiation	`2 ** 3`	`8`	Raises the first value to the power of the second.

---

6. Increment/Decrement Operators

Used to increase or decrease a value by 1.

Operator	Description	Example	Result	Explanation
`++`	Increment	`a++`	`a + 1`	Increases the value of a variable by 1.
`--`	Decrement	`a--`	`a - 1`	Decreases the value of a variable by 1.

---

3. Comparison Operators

Used to compare two values.

Operator	Description	Example	Result	Explanation
`==`	Equal to	`5 == '5'`	`true`	Compares values for equality (ignores type).
`===`	Strictly equal to	`5 === '5'`	`false`	Compares both value and type for equality.
`!=`	Not equal to	`5 != 3`	`true`	Checks if values are not equal (ignores type).
`!==`	Strictly not equal to	`5 !== '5'`	`true`	Checks if values and types are not equal.
`>`	Greater than	`5 > 3`	`true`	Returns `true` if the left value is greater than the right.
`<`	Less than	`5 < 3`	`false`	Returns `true` if the left value is less than the right.
`>=`	Greater than or equal to	`5 >= 5`	`true`	Returns `true` if the left value is greater than or equal to the right.
`<=`	Less than or equal to	`3 <= 5`	`true`	Returns `true` if the left value is less than or equal to the right.

7. Bitwise Operators

Operate at the bit level.

Operator	Description	Example	Result	Explanation
`&`	AND	`5 & 1`	`1`	Performs a bitwise AND operation.
`` ` ``	`` ` ``	OR	`5	1`	
`^`	XOR	`5 ^ 1`	`4`	Performs a bitwise XOR operation.
`~`	NOT	`~5`	`-6`	Inverts all the bits of the number.
`<<`	Left shift	`5 << 1`	`10`	Shifts bits to the left, filling with zeros.
`>>`	Right shift	`5 >> 1`	`2`	Shifts bits to the right, discarding bits shifted out.

2. Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Result	Explanation
`=`	Assign	`a = 5`	`5`	Assigns the value on the right to the variable on the left.
`+=`	Add and assign	`a += 3`	`8`	Adds the right operand to the left operand and assigns the result to the left operand.
`-=`	Subtract and assign	`a -= 2`	`3`	Subtracts the right operand from the left operand and assigns the result to the left operand.
`*=`	Multiply and assign	`a *= 2`	`10`	Multiplies the left operand by the right operand and assigns the result.
`/=`	Divide and assign	`a /= 2`	`2.5`	Divides the left operand by the right operand and assigns the result.

4. Logical Operators

Used for logical operations.

Operator	Description	Example	Result	Explanation
`&&`	Logical AND	`true && false`	`false`	Returns `true` if both conditions are true.
`	`		`	Logical OR	`true
`!`	Logical NOT	`!true`	`false`	Reverses the logical state of its operand.

---

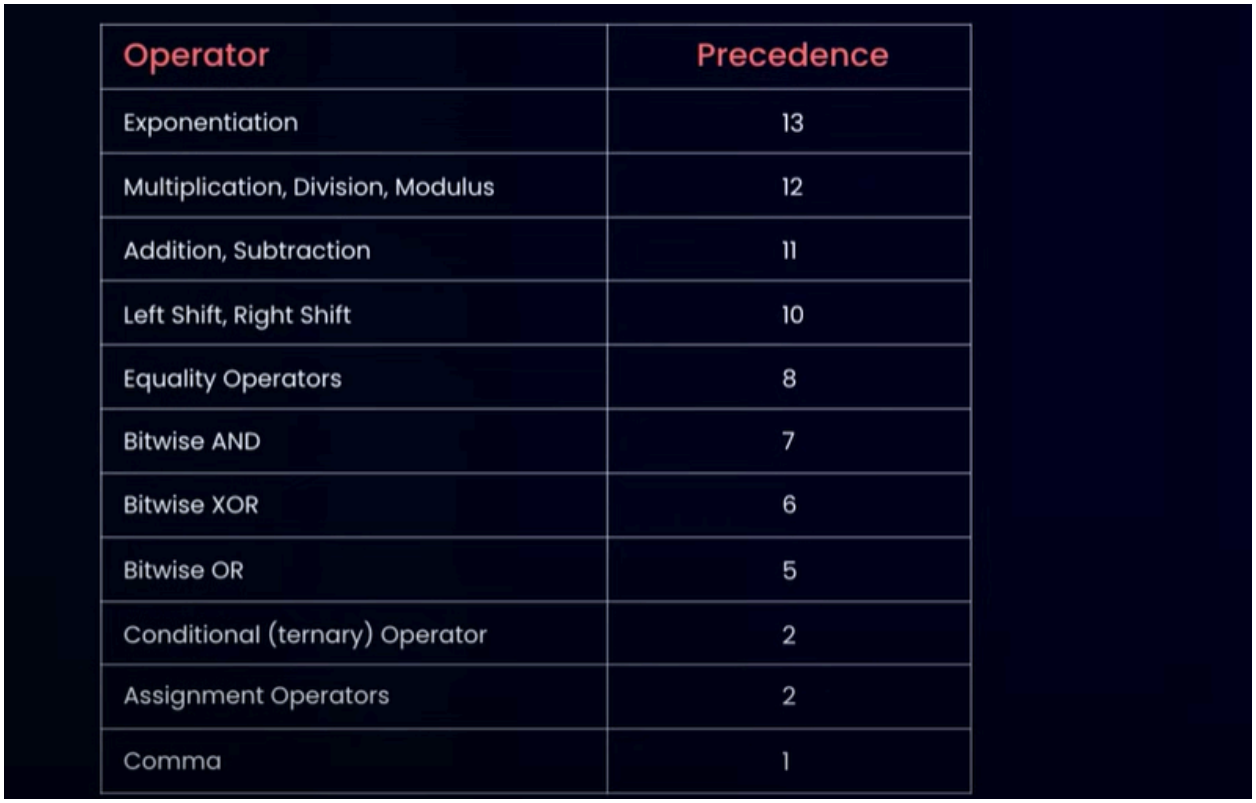
5. Conditional (Ternary) Operator

Shortcut for `if-else`.

Syntax	Description	Example	Result	Explanation
`condition ? expr1 : expr2`	If `condition` is true, return `expr1`, otherwise `expr2`.	`age >= 18 ? "Adult" : "Minor"`	`"Adult"`	Checks a condition and returns one of two values.

---

operator precedence in javascript



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

Conditional statements in javascript

1. if Statement

Executes a block of code if a specified condition is true.

```
if (condition) {
  // code to execute if condition is true
}
```

Example:

```
let age = 18;
if (age >= 18) {
  console.log("You are eligible to vote.");
}
```

---

2. if...else Statement

Executes one block of code if the condition is true and another if it is false.

```
if (condition) {
  // code to execute if condition is true
} else {
  // code to execute if condition is false
}
```

Example:

```
let isMember = true;
if (isMember) {
  console.log("Welcome, member!");
} else {
  console.log("Please sign up to become a member.");
}
```

---

3. if...else if...else Statement

Tests multiple conditions. Executes the first block of code where the condition is true.

```
if (condition1) {
  // code to execute if condition1 is true
```

```
} else if (condition2) {
  // code to execute if condition2 is true
} else {
  // code to execute if none of the conditions are true
}
```

Example:

```
let score = 85;
if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else {
  console.log("Grade: C");
}
```

---

4. Switch Statement

Used for multiple conditions. It's cleaner than multiple if...else if statements in some cases.

```
switch (expression) {
  case value1:
    // code to execute if expression === value1
```

```
    break;
  case value2:
    // code to execute if expression === value2
    break;
  default:
    // code to execute if no case matches
}
```

Example:

```
let day = 3;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  default:
    console.log("Other day");
}
```

---

5. Ternary Operator

A compact form of if...else.

```
condition ? expressionIfTrue : expressionIfFalse;
```

Example:

```
let age = 20;
let eligibility = age >= 18 ? "Eligible to vote" : "Not eligible to vote";
console.log(eligibility);
```

---

6. Logical Operators in Conditions

You can combine conditions using logical operators like && (AND), || (OR), and ! (NOT).

Example with &&:

```
let age = 25;
if (age > 18 && age < 30) {
  console.log("You are a young adult.");
}
```

Example with ||:

```
let day = "Saturday";
if (day === "Saturday" || day === "Sunday") {
```

```
  console.log("It's the weekend!");
}
```

Example with !:

```
let isRaining = false;
if (!isRaining) {
  console.log("You can go outside without an umbrella.");
}
```

```
// Example: Weather and Outfit Recommendation

let weather = "sunny";  // Change this to "rainy", "cloudy", or other values
let temperature = 30;   // Temperature in Celsius
let isWeekend = true;

// 1. Using 'if'
if (weather === "sunny") {
  console.log("It's a sunny day! Wear light clothes.");
}

// 2. Using 'if...else'
if (weather === "rainy") {
  console.log("It's rainy! Don't forget your umbrella.");
} else {
```

```
  console.log("No rain today. Enjoy!");
}

// 3. Using 'if...else if...else'
if (temperature > 35) {
  console.log("It's extremely hot outside. Stay hydrated!");
} else if (temperature > 25) {
  console.log("It's warm. A perfect day for a walk.");
} else if (temperature > 15) {
  console.log("It's cool. A light jacket will do.");
} else {
  console.log("It's cold. Wear warm clothes.");
}

// 4. Using 'switch'
switch (weather) {
  case "sunny":
    console.log("Great weather for outdoor activities.");
    break;
  case "rainy":
    console.log("Better to stay indoors or wear a raincoat.");
    break;
  case "cloudy":
```

```
    console.log("It might rain later, stay prepared.");
    break;
  default:
    console.log("Weather is unpredictable today. Stay alert!");
}

// 5. Using ternary operator
let outingPlan = isWeekend
  ? "It's the weekend! Plan a trip."
  : "It's a weekday. Focus on work or school.";
console.log(outingPlan);

// 6. Using logical operators
if (weather === "sunny" && temperature > 25) {
  console.log("Perfect day for the beach!");
} else if (weather === "rainy" || temperature < 10) {
  console.log("Not ideal for outdoor activities. Maybe read a book?");
}

if (!isWeekend) {
  console.log("Time to hustle!");
} else {
  console.log("Relax and unwind!");
```

```
}
```

Output

```
It's a sunny day! Wear light clothes.                    script2.js:9
No rain today. Enjoy!                                    script2.js:16
It's warm. A perfect day for a walk.                     script2.js:23
Great weather for outdoor activities.                    script2.js:33
It's the weekend! Plan a trip.                           script2.js:49
Perfect day for the beach!                               script2.js:53
Relax and unwind!                                        script2.js:61
Live reload enabled.                                     Home.html:48
```

operators

In JavaScript, == **(equality operator)** and === **(strict equality operator)** are used to compare values, but they differ in how they handle type conversion.

---

1. == (Equality Operator)

- **Type Conversion:** Performs type coercion (implicit conversion) if the types of the operands are different.
- **Comparison:** Compares the values after converting them to a common type.

Examples:

```
console.log(5 == "5");   // true (string "5" is converted to number 5)

console.log(0 == false); // true (false is converted to number 0)

console.log(null == undefined); // true (special case in JavaScript)

console.log(" " == 0);   // true (empty string is converted to number 0)
```

Pitfalls:

- Can lead to unexpected results due to type coercion.

```
console.log("5" == true); // false ("5" is converted to number, but true is 1)

console.log([] == false); // true (empty array converts to 0)
```

---

2. === (Strict Equality Operator)

- **Type Conversion:** Does not perform type coercion.
- **Comparison:** Compares both value and type strictly.

Examples:

```
console.log(5 === "5");   // false (different types)

console.log(0 === false); // false (different types)

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

console.log(" " === 0);   // false (different types)
```

Advantages:

- Ensures clarity and avoids unexpected behavior caused by type coercion.

Switch Statement

A **switch statement** in JavaScript is used for decision-making based on multiple conditions. It provides a cleaner and more readable alternative to using multiple if-else statements when comparing a single variable or expression against multiple possible values.

```
switch (expression) {

  case value1:
```

```
    // Code to execute if expression === value1
    break;
  case value2:
    // Code to execute if expression === value2
    break;
  default:
    // Code to execute if no case matches
}
```

```
let fruit = "apple";
switch (fruit) {
  case "apple":
    console.log("Apples are $1 each.");
    break;
  case "banana":
    console.log("Bananas are $0.5 each.");
    break;
  case "cherry":
    console.log("Cherries are $3 per pound.");
    break;
  default:
    console.log("Sorry, we don't have that fruit.");
}
```

Loops

Loops in JavaScript are used to execute a block of code repeatedly until a specified condition is met. JavaScript provides several types of loops:

1. For Loop

Used when you know the exact number of iterations.

Syntax:

```
for (initialization; condition; increment/decrement) {
  // Code to execute
}
```

Example:

```
// Print numbers from 1 to 5
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

Output:

```
1
2
3
4
5
```

2. While Loop

Used when the number of iterations is not known beforehand, but the condition is.

Syntax:

```
while (condition) {
  // Code to execute
}
```

Example:

```
// Print numbers from 1 to 5
let i = 1;
while (i <= 5) {
  console.log(i);
  i++;
}
```

Output:

```
1
2
3
4
5
```

3. Do-While Loop

Executes the block of code first before checking the condition.

Syntax:

```
do {
  // Code to execute
```

```
} while (condition);
```

Example:

```
// Print numbers from 1 to 5

let i = 7;

do {

  console.log(i);

  i++;

} while (i <= 5);
```

Output:

```
7
```

4. For...of Loop

Iterates over iterable objects (e.g., arrays, strings).

The for...of loop iterates over **iterable objects**, such as:

- Arrays
- Strings
- Sets
- Maps
- Typed Arrays
- Other iterable objects

It returns the **values** of the iterable.

Syntax:

```
for (variable of iterable) {

  // Code to execute
```

}

Example:

```
// Iterate through an array

let colors = ['red', 'green', 'blue'];

for (let color of colors) {

  console.log(color);

}
```

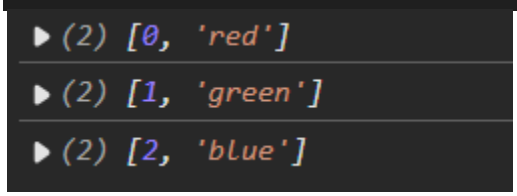
Output:

red

green

Blue

```
for (let color of colors.entries()) {

  console.log(color);

}
```



When to Use Which?

- Use `for`:
  - When you need access to both the index and value.
  - When you want fine-grained control over the iteration (e.g., custom step sizes, stopping early, etc.).
  - When iterating over non-iterable data structures.
- Use `for...of`:
  - When you only care about the values, not the indexes.
  - When working with iterable objects (arrays, strings, sets, maps, etc.).
  - For cleaner and more readable code, especially in modern JavaScript.

5. For...in Loop

Iterates over the enumerable properties of an object.

The for...in loop iterates over the **enumerable properties** of an object. It is mainly used with objects, though it can also work with arrays (less common and not recommended).

It returns the **keys** (property names or array indexes).

Syntax:

```
for (key in object) {
  // Code to execute
}
```

Example:

```
// Iterate through an object's properties
let person = { name: 'Alice', age: 25 };
for (let key in person) {
  console.log(`${key}: ${person[key]}`);
```

```
}
```

Output:

```
name: Alice

age: 25
```

6. Break and Continue

- **break**: Exits the loop immediately.
- **continue**: Skips the current iteration and continues with the next one.

Example:

```
// Break example: Stop at 3

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

  if (i === 3) break;

  console.log(i);

}

// Continue example: Skip 3

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

  if (i === 3) continue;

  console.log(i);

}
```

Output:

For break:

```
1

2
```

For continue:

```
1
```

```
2
4
5
```

---

7. Nested Loops

Loops inside another loop.

Example:

```
// Print a 3x3 grid
for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 3; j++) {
    console.log(`Row ${i}, Column ${j}`);
  }
}
```

Output:

```
Row 1, Column 1
Row 1, Column 2
Row 1, Column 3
Row 2, Column 1
Row 2, Column 2
Row 2, Column 3
Row 3, Column 1
Row 3, Column 2
```

Row 3, Column 3

JavaScript Functions

JavaScript functions are used to perform operations. We can call JavaScript functions many times to reuse the code.

Advantage of JavaScript function

Functions are useful in organizing the different parts of a script into the several tasks that must be completed. There are mainly two advantages of JavaScript functions.

1. Code reusability: We can call a function several times in a script to perform their tasks so it saves coding.
2. Less coding: It makes our program compact. We don't need to write many lines of code each time to perform a common task.

Rules for naming functions:

- It must be case sensitive.
- It must start with an alphabetical character (A-Z) or an underscore symbol.
- It cannot contain spaces.
- It cannot be used as a reserve word.

How to declare a Function:

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: {}

```
function name(parameter1, parameter2, parameter3) {


  // code to be executed
```

}

1) Default Function (Function Declaration)

A **default function** is declared using the function keyword followed by the function name.

Syntax:

```
function functionName(parameters) {
  // function body
}
```

Example:

```
function greet(name) {
  console.log("Hello, " + name + "!");
}
greet("Alice"); // Output: Hello, Alice!
```

2) Function with a Return Value

A function can return a value using the return keyword.

Example:

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


let result = add(5, 7);
console.log(result); // Output: 12
```

3) Default Parameters in Functions

You can set default values for function parameters.

Example:

```
function greet(name = "Guest") {
  return "Hello, " + name + "!";
}
console.log(greet());       // Output: Hello, Guest!
console.log(greet("Alice")); // Output: Hello, Alice!
```

2) Anonymous Function/ Function expression

An anonymous function is simply a function that does **not have a name**

An **anonymous function** is a function without a name. It is often used as a value for variables or passed as arguments.

Syntax

The below-enlightened syntax illustrates the declaration of an anonymous function using the normal declaration:

```
function() {
  // Function Body
}
```

We may also declare an anonymous function using the arrow function technique which is shown below:

```
( () => {
  // Function Body...
} )();
```

Example:

```
let greet = function(name) {
  return "Hello, " + name + "!";
};
```

```
console.log(greet("Bob")); // Output: Hello, Bob!
```

Passing arguments to the anonymous function.

```
const greet = function( str ) {
    console.log("Welcome to ", str);
};
greet("GeeksforGeeks!");
```

Creating a self-executing function.

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

2) Passing an anonymous function as a callback function to the *setTimeout()* method. This executes this anonymous function 2000ms later.

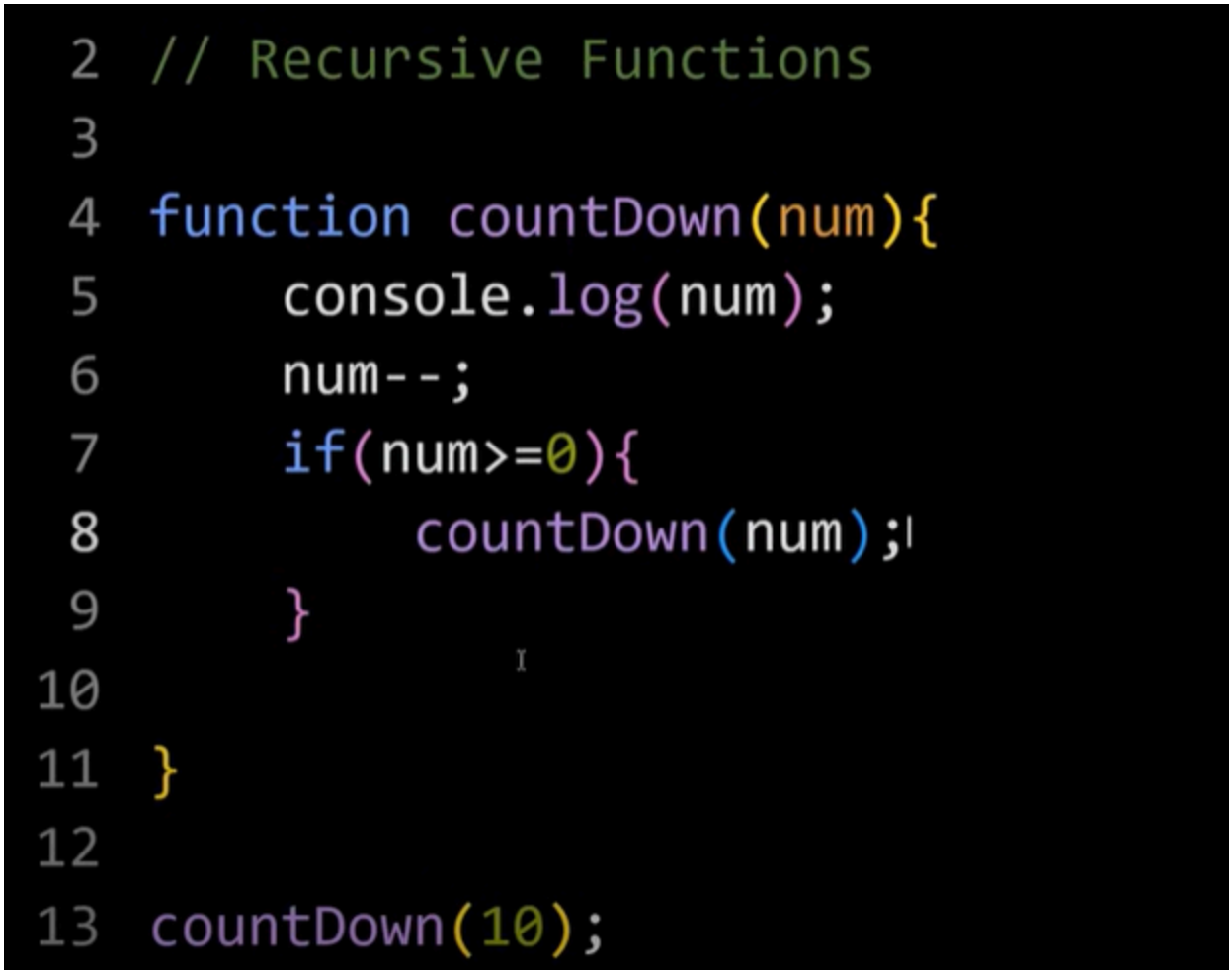
```
setTimeout(function () {
    console.log("Welcome to GeeksforGeeks!");
}, 2000);
```

Output

Welcome to GeeksforGeeks!

3) Recursive Function

A recursive function is a function that calls itself.



```
// Recursive Functions

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

}

countDown(10);
```

3) Arrow Function(lambda functions in some other programming languages)

An arrow function is essentially an anonymous function with a shorter syntax. They are often assigned to variables, making them reusable. Arrow functions are also known as lambda functions in some other programming languages.

ES6 introduced the Arrow functions in JavaScript which offer a more concise and readable way to write function expressions.

Using functions expression/anonymous function will achieve arrow function

Syntax

```
const gfg = () => {
```

```
    console.log( "Hi Geek!" );
}
```

The below examples show the working of the Arrow functions in JavaScript.

1. Arrow Function without Parameters
An arrow function without parameters is defined using empty parentheses (). This is useful when you need a function that doesn't require any arguments.

Example: In this example we define an arrow function gfg without parameters that logs "Hi from GeekforGeeks!" when called.

```
const gfg = () => {
    console.log( "Hi from GeekforGeeks!" );
}
gfg();
```

Output

```
Hi from GeekforGeeks!
```

2. Arrow Function with Single Parameters
If your arrow function has a single parameter, you can omit the parentheses around it.

Example: In this example we defines an arrow function square with a single parameter x, returning the square of x.

```
const square = x => x*x;
console.log(square(4));
// output: 16
```

Output

```
16
```

3. Arrow Function with Multiple Parameters

Arrow functions with multiple parameters, like **(param1, param2) => { }**, simplify writing concise function expressions in JavaScript, useful for functions requiring more than one argument.

Example : In this example we defines an arrow function gfg with parameters x, y, z, logging their sum.

```
const gfg = ( x, y, z ) => {

    console.log( x + y + z )

}

gfg( 10, 20, 30 );
```

Output

```
60
```

4. Arrow Function with Default Parameters

Arrow functions support default parameters, allowing predefined values if no argument is passed, making JavaScript function definitions more flexible and concise.

Example : In this example we define an arrow function gfg with parameters x, y, and a default parameter z = 30.

```
const gfg = ( x, y, z = 30 ) => {

    console.log( x + " " + y + " " + z);

}

gfg( 10, 20 );
```

Output

```
10 20 30
```

//Function calling other function

```
function add(a, b) {
   return a + b;
}
function calculateAndPrint() {
   const result = add(5, 7); // Call `add` with arguments
   console.log(`The result is: ${result}`);
}
calculateAndPrint(); // Outputs: The result is: 12
```

Closer Look at function

1)Default parameters

```
'use strict';



const bookings = [];

const createBooking = function (flightNum, numPassengers = 1, price =
1000) {

  const booking = {

    flightNum,

    numPassengers,

    price,

  };

  console.log(booking);

  bookings.push(booking);

};
```

```
createBooking('LH123', 2, 12000); //{flightNum: 'LH123', numPassengers: 2,
price: 12000}

createBooking('LH124', undefined, 12000); //{flightNum: 'LH124',
numPassengers: 1, price: 12000}

createBooking('LH125'); //{flightNum: 'LH125', numPassengers: 1, price:
1000}
```

2) values vs ref

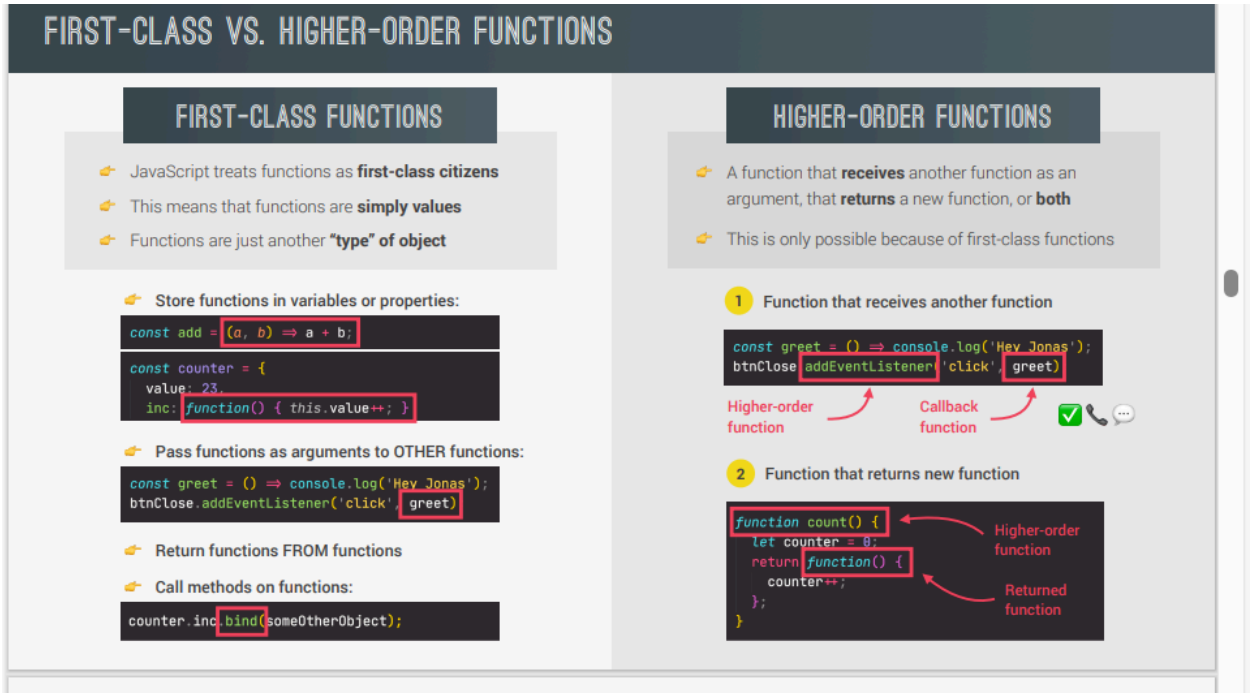
What is “Pass by Value” in JavaScript?
Pass by Value means that when you pass a variable to a function, JavaScript creates a copy of the variable’s value and uses it inside the function. This means any changes made to the variable inside the function do not affect the original variable outside the function.

```
function changeValue(x) {

  x = 10;

  console.log(x); // 10

}

let a = 5;

changeValue(a);

console.log(a); // 5 (original value of a is unchanged)
```

What is “Pass by Reference” in JavaScript?
Pass by Reference means that when you pass a variable (specifically, objects or arrays) to a function, JavaScript passes the reference or memory address of the variable, not a copy. This means any changes made to the variable inside the function will affect the original variable outside the function.

```
// Object passed by reference (value of reference)

function modifyObject(obj) {
```

```
  obj.name = 'dhoni'; // Modify the property of the object

  obj = { name: 'king' }; // Change the reference (doesn't affect the
original object)

}

let person = { name: 'ram', age: 20 };

console.log(person); //{name: 'ram', age: 20}

modifyObject(person);

console.log(person); //{name: 'dhoni', age: 20} (object is modified, but
reference change does not affect outside)
```



Example:argument vs parameter

// Function definition (with parameters)

function multiply(a, b) {

```
  return a * b;
}

// Function call (with arguments)
multiply(4, 5);  // 4 and 5 are arguments
```

JavaScript callback

A callback function can be defined as a function passed into another function as a parameter. Don't relate the callback with the keyword, as the callback is just a name of an argument that is passed to a function.

In other words, we can say that **a function passed to another function as an argument is referred to as a callback function.**

Synchronous Callback

Here's an example where a synchronous callback is used to add two numbers:

Don't relate the callback with the keyword, as the callback is just a name of an argument that is passed to a function

```
function add(a, b, callback) {
  let result = a + b;
  callback(result);  // Calling the callback function
}

function displayResult(result) {
  console.log("The result is: " + result);
}

// Using the add function with displayResult as a callback
add(5, 3, displayResult); // Output: The result is: 8
```

```
function add(a, b, fn) {
let result = a + b;
fn(result); // Calling the callback function
}

function displayResult(result) {
  console.log('The result is: ' + result);
}
// Using the add function with displayResult as a callback
add(5, 3, displayResult); // Output: The result is: 8
```


The function add takes two numbers and a callback function (displayResult).

After adding the numbers, it calls displayResult to display the result.

The displayResult function runs **synchronously** once the addition is complete.

Higher order function

JavaScript Higher-Order Functions are functions that can accept other functions as arguments, return functions, or both. They enable abstraction and flexibility in code, allowing you to create reusable and modular functions for complex operations, making them essential in functional programming.

Function that receives another function

```
function greet(name) {

  return `Hello, ${name}!`; //// Output: Hello, ram!

}

function displayGreeting(fn, name) {

  console.log(fn(name)); // The function greet is passed as an argument

}

displayGreeting(greet, 'ram');
```

Function returning function

```
const greet = function (message) {

  return function (input) {

    console.log(`The ${message} has received at ${input} minutes ago `);

  };
```

```
};

const mes = greet('Good Morning');

console.log(mes); // it will return functiondef ƒ (input)
{console.log(`The${message} has received at ${input}` minutes ago); }

mes(23); // pass arguments to inner function then will get output

//greet('Good Morning')('23'); will get same result
```

Call , apply and bind methods

The **call() method** is a predefined JavaScript method. It can be used to invoke (call) a method with an owner object as an argument (parameter). This allows borrowing methods from other objects, executing them within a different context, overriding the default value, and passing arguments.

The call() method calls a function with a specified this value and arguments provided individually.

Syntax:

object.objectMethod.call(objectInstance, arguments)

functionName.call(thisArg, arg1, arg2, ...);

```
const employee = {

    details: function (designation, experience) {

      return this.name + ' ' + this.id + designation + experience;

    },

  };

  // Objects declaration
```

```
const emp1 = {

  name: 'A',

  id: '123',

};

const emp2 = {

  name: 'B',

  id: '456',

};

const x = employee.details.call(emp2, ' Manager ', '4 years');

console.log(x);
```

Output

B 456 Manager 4 years

apply()

The apply() method in JavaScript is a built-in function of Function objects that allows you to call a function with a specific this value and arguments provided as an **array (or an array-like object)**.

The apply method is similar to call, but it takes arguments as an array or an array-like object

Syntax:

apply(objectInstance)

apply(objectInstance, argsArray)

function.apply(thisArg, [argsArray])

Ex: the apply() function without arguments.

```
let student = {
    details: function () {
        return this.name + this.class;
    }
}
let stud1 = {
    name: "Dinesh",
    class: "11th",
}
let stud2 = {
    name: "Vaibhav",
    class: "11th",
}

let x = student.details.apply(stud2);
console.log(x);
```

```
Vaibhav
11th
```

Ex: apply() function with arguments.

```
let student = {
    details: function (section, rollnum) {
        return this.name + this.class
            + " " + section + rollnum;
    }
}
let stud1 = {
    name: "Dinesh",
    class: "11th",
}
let stud2 = {
    name: "Vaibhav",
    class: "11th",
}

let x = student.details.apply(stud2, ["A", "24"]);
console.log(x);
```

Output:

```
Vaibhav
11th A
```

24

Bind()

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

It doesn't immediately call a function immediately instead it will return a new function.

```
func.bind(thisArg, arg1, ... argN)
```

Using bind() Method without parameters

```
const person = {
  firstName: 'ravi',
  lastName: 'D',
  fullName: function () {
    return this.firstName + ' ' + this.lastName;
  },
};
console.log(person.fullName()); //ravi D

const member = {
  firstName: 'raju',
  lastName: 'k',
};

// Bind `member` to `fullName` method of `person`
let fullName = person.fullName.bind(member);

console.log(fullName());
// Output: raju k
```

Using bind() Method with parameters

```
const college1 = {

  collegeName: 'Mrec',

  group: 'eee',

  area: 'hyd',
```

```
  btech(admission, name) {

    console.log(

      `${name} has joined in ${this.group} at
${this.collegeName},${this.area} on ${admission} `

    );

  },

};

college1.btech('2024-12-12', 'ram');

college1.btech('2024-12-12', 'dhoni');



//College2

const college2 = {

  collegeName: 'NITWGL',

  group: 'CSE',

  area: 'WGL',

};



const usingBind = college1.btech.bind(college2, '2024-12-13', 'king');

usingBind();
```

```
ram has joined in eee at Mrec,hyd on 2024-12-12
dhoni has joined in eee at Mrec,hyd on 2024-12-12
king has joined in CSE at NITWGL,WGL on 2024-12-13
```

Ex ;Scenario: Movie Ticket Booking

We have a movieTicket object representing a booking system for a specific theater chain. We'll reuse the book function for other chains using the call and apply methods.

```
const Asion = {

  theater: 'IMAX',

  chainCode: 'IMX',

  bookings: [],

  //book: function(){}

  book(ticketNum, customerName) {

    console.log(

      `${customerName} booked ticket #${ticketNum} at ${this.theater}
(${this.chainCode})`

    );

    this.bookings.push({

      ticket: `${this.chainCode}${ticketNum}`,

      customer: customerName,

    });

  },

};
```

```
// Booking directly on the original `movieTicket` object

Asion.book(101, 'ram');

Asion.book(102, 'dhoni');

console.log(Asion.bookings);



// other theaters



const AMB = {

  theater: 'AMB HYD',

  chainCode: 'MB',

  bookings: [],

};



const cineplex = {

  theater: 'Cineplex',

  chainCode: 'CPL',

  bookings: [],

};



// Reusing the `book` function in Asion  for other theaters like AMB and
cineplex



// 1.Assigning the `book` function to a standalone variable

const book = Asion.book;
```

```
// Using `call` to book tickets on `AMB`

book.call(AMB, 201, 'kl');

book.call(AMB, 202, 'virat');

console.log(AMB.bookings);

// Using `call` to book tickets on `cineplex`

book.call(cineplex, 301, 'rishab');

console.log(cineplex.bookings);



// Using `apply` to book tickets on `cineplex` with an array of arguments

const ticketDetails = [302, 'vinay'];

book.apply(AMB, ticketDetails);

console.log(AMB.bookings);



// Using spread syntax as a modern alternative to `apply`

book.call(cineplex, ...ticketDetails);

console.log(cineplex.bookings);

//bind

const bindBook = book.bind(AMB, 204, 'vinnu');

bindBook();

console.log(AMB.bookings);
```

Output

ram booked ticket #101 at IMAX (IMX)

dhoni booked ticket #102 at IMAX (IMX)

[{ ticket: 'IMX101', customer: 'ram' }, { ticket: 'IMX102', customer: 'dhoni' }]

kl booked ticket #201 at AMB HYD (MB)

virat booked ticket #202 at AMB HYD (MB)

[{ ticket: 'MB201', customer: 'kl' }, { ticket: 'MB202', customer: 'virat' }]

rishab booked ticket #301 at Cineplex (CPL)

[{ ticket: 'CPL301', customer: 'rishab' }]

vinay booked ticket #203 at AMB HYD (MB)

[{ ticket: 'MB201', customer: 'kl' }, { ticket: 'MB202', customer: 'virat' }, { ticket: 'MB203', customer: 'vinay' }]

vinay booked ticket #203 at Cineplex (CPL)

[{ ticket: 'CPL301', customer: 'rishab' }, { ticket: 'CPL203', customer: 'vinay' }]

[{ ticket: 'MB201', customer: 'kl' }, { ticket: 'MB202', customer: 'virat' }, { ticket: 'MB203', customer: 'vinay' }]

Coding challenge

Let's build a simple poll app! A poll has a question, an array of options from which people can choose, and an array with the number of replies for each option. This data is stored in the starter 'poll' object below.

Your tasks:

1. Create a method called 'registerNewAnswer' on the 'poll' object.

The method does 2 things:

1.1. Display a prompt window for the user to input the number of the selected option. The prompt should look like this:

What is your favourite programming language?

0: JavaScript

1: Python

2: Rust

3: C++ (Write option number)

1.2. Based on the input number, update the 'answers' array property. For example, if the option is 3, increase the value at position 3 of the array by

1. Make sure to check if the input is a number and if the number makes sense (e.g. answer 52 wouldn't make sense, right?)

2. Call this method whenever the user clicks the "Answer poll" button.

3. Create a method 'displayResults' which displays the poll results. The method takes a string as an input (called 'type'), which can be either 'string' or 'array'. If type is 'array', simply display the results array as it is, using console.log(). This should be the default option. If type is 'string', display a string like "Poll results are 13, 2, 4, 1".

4. Run the 'displayResults' method at the end of each 'registerNewAnswer' method call.

5. Bonus: Use the 'displayResults' method to display the 2 arrays in the test data. Use both the 'array' and the 'string' option. Do not put the arrays in the poll object! So what should the this keyword look like in this situation? The Complete JavaScript Course 21

Test data for bonus:

Data 1: [5, 2, 3]

Data 2: [1, 5, 3, 9, 6, 1]

Hints: Use many of the tools you learned about in this and the last section

```
//Challenge

const poll = {

  question: 'What is your favourite programming language?',

  options: ['0: JavaScript', '1: Python', '2: Rust', '3: C++'],

  // This generates [0, 0, 0, 0]. More in the next section!

  answers: new Array(4).fill(0),

  registerNewAnswer() {

    //get answer

    const answer = Number(
```

```
      prompt(
        `${this.question}\n${this.options.join('\n')}\n(Write option
number)`
      )
    );
    console.log(answer);
    //register answer
    typeof answer === 'number' &&
      answer < this.answers.length &&
      this.answers[answer]++;
    console.log(this.answers);
  },
  displayResults(type) {
    if (type === 'array') {
      console.log(answers);
    } else if (type === 'string')
      console.log(`Poll results are ${this.answers.join(',')}`);
  },
};
//poll.registerNewAnswer();

document
  .querySelector('.poll')
  .addEventListener('click', poll.registerNewAnswer.bind(poll));
```

```
poll.displayResults.call({ answers: [5, 2, 3] }, 'string'); //Poll results
are 5,2,3

poll.displayResults.call({ answers: [1, 5, 3, 9, 6, 1] }, 'string');
//Poll results are 1,5,3,9,6,1
```

Immediately Invoked Function expression

An **Immediately Invoked Function Expression** (IIFE) is a JavaScript function that is defined and immediately executed as soon as it is defined.

An IIFE is a function that is invoked immediately after being defined.

It's used to create a local scope, avoid global variable pollution, and can encapsulate code.

It’s a common pattern in JavaScript, especially before the introduction of ES6 modules.

Syntax:

```
(function() {
  // Code to be executed immediately
})();
```

Or

with an arrow function:

```
(() => {
  // Code to be executed immediately
})();
```

Ex1

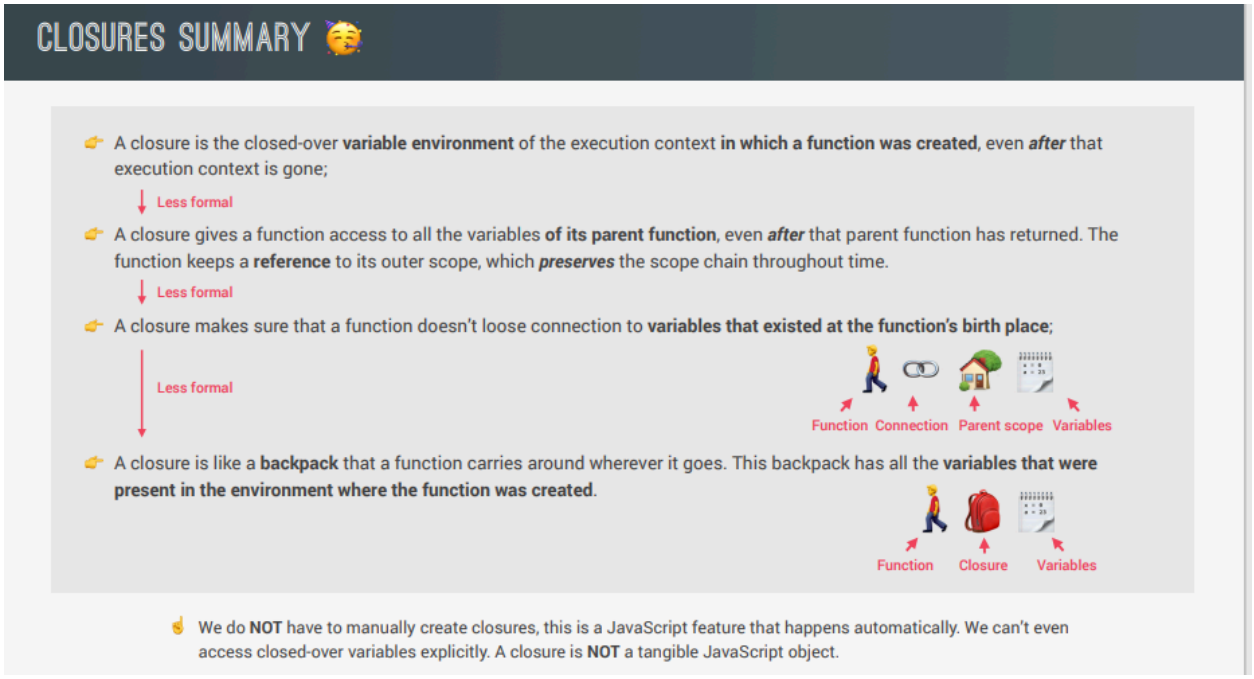
```
(function(a, b) {
```

```
  console.log(a + b); // Output: 5
})(2, 3);
```

Ex2

```
(function() {
  var name = "Alice";
  console.log("Hello, " + name); // Output: Hello, Alice
})();
```

Closures



A closure is a function that **"remembers" its lexical scope**, **even when the function is executed outside that scope**. This means that a closure has access to variables from its outer function even after the outer function has finished executing.

Ex1:

```
//closure

function init() {

  var name = 'Ramesh'; // name is a local variable created by init
```

```
  function displayName() {

    // displayName() is the inner function, that forms a closure

    console.log(name); // use variable declared in the parent function

  }

  displayName();

}

init();
```

Output

Ramesh

init() creates a local variable called name and a function called displayName(). The displayName() function is an inner function that is defined inside init() and is available only within the body of the init() function. Note that the displayName() function has no local variables of its own. However, since inner functions have access to the variables of outer scopes, displayName() can access the variable name declared in the parent function, init().

Step-by-Step Execution of the Code

1. Global Execution Context (GEC)

- The init function is stored in memory.
- No other variables are defined globally.

2. init() is Invoked

- A **Function Execution Context (FEC)** for init is created.
    - **Creation Phase**:
        - The variable name is allocated memory and initialized with the value 'Ramesh'.
        - The function displayName is stored in memory.
    - **Execution Phase**:
        - The variable name is set to 'Ramesh'.
        - The displayName() function is called.

3. displayName() is Invoked

- A new **FEC** for displayName is created.
    - **Creation Phase**:
        - No local variables or parameters are declared in displayName, so its local memory space remains empty.
    - **Execution Phase**:
        - The console.log(name) statement is executed.
        - JavaScript looks for the variable name in the local scope of displayName. It doesn't find it there, so it moves to the outer (parent) scope, which is init().
        - The value 'Ramesh' is found in the init function's scope and logged to the console.

Understanding Closure

A **closure** is formed when a function "remembers" variables from its lexical scope, even after the parent function has finished execution.

- In this code, displayName is a **closure** because it remembers the variable name from the init function's scope, even though init has finished execution when displayName is invoked.

EX2:

```
let sample;

const functionFor1 = function () {

  const a = 23;

  sample = function () {

    console.log(a * 2);

  };

};

const functionFor2 = function () {

  const b = 777;

  sample = function () {

    console.log(b * 2);
```

```
  };

};

functionFor1();

sample();

console.dir(sample);

// Re-assigning sample function

functionFor2();

sample();

console.dir(sample);
```

Output

Detailed Explanation

Step 1: Variables and Functions

1. sample is declared as a let variable but not initialized. It will later store functions dynamically.
2. Two functions, functionFor1 and functionFor2, are defined:
    - Both functions assign a new function to sample.
    - These inner functions have access to variables (a or b) declared in their respective outer functions, creating a closure.

Step 2: functionFor1 Execution

```
functionFor1();
```

- When functionFor1 is called:
    - A local variable a is declared and initialized with the value 23.

```
sample is assigned a function:
sample = function () {

  console.log(a * 2);

};
```

    - This new function forms a closure over the variable a.

Step 3: Calling sample

```
sample();
```

- When sample() is invoked, it logs a * 2:
    - The sample function has access to the a variable from functionFor1 (due to closure).
    - Output: 46 (since 23×2=4623 \times 2 = 4623×2=46).

Step 4: Inspecting sample

```
console.dir(sample);
```

- console.dir(sample) outputs details about the sample function, showing that it is a closure.
- It reveals the internal structure of sample, including:
    - The code for sample.
    - The closure, showing the captured variable a and its value (23).

Step 5: functionFor2 Execution

functionFor2();

- When functionFor2 is called:
    - A local variable b is declared and initialized with the value 777.

sample is re-assigned to a new function:
sample = function () {

console.log(b * 2);

};

    - The previous function stored in sample (from functionFor1) is replaced.
    - This new function forms a closure over the variable b.

Step 6: Calling sample Again

sample();

- When sample() is invoked, it logs b * 2:
    - The sample function now has access to the b variable from functionFor2.
    - Output: 1554 (since 777×2=1554777 \times 2 = 1554777×2=1554).

Step 7: Inspecting sample Again

console.dir(sample);

- console.dir(sample) now reveals the details of the new sample function.
- It shows the new closure:
    - The captured variable b and its value (777).

Key Concepts Demonstrated

1. **Closure**:
    - Each function assigned to a sample retains access to its respective outer variables (a or b), even after functionFor1 or functionFor2 has finished executing.
2. **Function Re-assignment**:
    - The sample variable is dynamically reassigned, replacing the previous closure with a new one.
3. **Lexical Scope**:
    - The inner functions retain their access to variables declared in the outer function where they were defined.

History

1. **Brendan Eich** created the very first version of JavaScript, called **Mocha**, in just 10 days in **1995**.
2. **Mocha** was later renamed to **LiveScript** and then to **JavaScript** in **1996** to attract Java developers.
3. **Microsoft** launched **Internet Explorer** and copied JavaScript from Netscape, calling it **JScript**.
4. **ECMAScript 1 (ES1)**, the first official version of the JavaScript standard, was released in **1997**.
5. **ECMAScript 5 (ES5)** was released in **2009** with many significant features, including strict mode and getter/setter support.
6. **ECMAScript 6 (ES6)**, also known as **ECMAScript 2015**, was released in **2015** and is considered the biggest update to the language ever, introducing features like arrow functions, classes, and promises.
7. To streamline the development process, **ECMAScript** adopted an annual release cycle, starting in **2016**, to ship smaller, more manageable feature updates each year.

The latest version of ECMAScript is ECMAScript 2024 (ES2024).

JavaScript is backward compatible. This means that older JavaScript code continues to work in newer versions of JavaScript without requiring modifications.

Old features are never removed: JavaScript maintains backward compatibility, ensuring that old features continue to work even with new versions.

Incremental updates: New versions of ECMAScript are not entirely new languages but rather incremental updates, adding new features and improvements without breaking existing code.

Websites keep working forever: Thanks to backward compatibility, websites built with older versions of JavaScript will continue to function as browsers and JavaScript engines evolve.

How to use Modern Javascript Today

1) During development use latest version of chrome
2) By using Babel along with polyfills, you can ensure your modern JavaScript or TypeScript code works across older browsers.

ECMAScript 6 (ES6)

Use let and const for Variable Declarations

Arrow Functions

Template Literals

Classes

…etc implemented

——————

```
/*



let str = "ramesh";

let salary = 3000;

console.log(str, salary);// we can also use a single console to print multiple values.

let age = 18;

if (age >= 18) {

    console.log("Eligible to vote"); // print data in console not on the page

} else {

    console.log(" Not Eligible to vote");

}

age = 10;//we can reassign values to same variable name but we can't initialize the value(let age)



//Data types

//primitive : number, string, boolean, undefined, null

//non-primitive: objects , array , functions ,date
```

```
//variable declaration

let danceRain = 'raining';

console.log(danceRain);

let DanceRain = 89.67; // case sensitive

console.log(DanceRain);

let dance_Rain = false;

console.log(dance_Rain);

let $dance = "rain"; // myVar$ ,

console.log($dance);

let r56 = 123;

console.log(r56);

let _gender = "male";

let val = null;

console.log(val);


//typeOf

console.log(_gender);

console.log(typeof (danceRain));

console.log(typeof (DanceRain));

console.log(typeof (dance_Rain));

console.log(typeof (r56));
```

```
console.log(typeof (val)); //object(exiting bug in javascript)



// 1) var is function-scoped, meaning it is accessible within the function where it is declared (if declared inside a function) or globally if declared outside any function.

var a = 10;

console.log(a);

var a = 90;

console.log(a); // we can reinitialize the same variable and value also;



console.log(b); // we can get undefined value without initializing ,printing first and initializing variables later

var b;

var c;

console.log(c); // we can get undefined value initializing variable first and printing value later



//2) let is block-scoped, meaning it is only available within the block (a pair of {}) where it is defined, such as within loops or conditionals.

//Preferred for Reassignable Variables

let name = "ramesh";

console.log(name);

name = 'rahul';

console.log(name);//we can reassign values to same variable name but we can't reinitialize the value(let name)

let lastName;

console.log(lastName);// we can get undefined value initializing variable first and printing value later
```

```
//console.log(firstName); //we can't get value,printing first and initializing variable  later

//let firstName;

let isIsland = false;

let language;

console.log(typeof isIsland);

console.log(typeof population);



//3) const:const cannot be redeclared within the same scope,  const does not allow reassignment after the initial assignment

const f = 123;

console.log(f);

//f = 90; we can't re assign

//const f=90;we can't re declare

//const t; value must be assigned to variable initially



*/



//Operators



/*

1) Arithmetic Operators(+, -, *,/, %, **)

2) Comparison (Relational) Operators(==,===,!=,!===,>,<,>=,<=)

3) Bitwise Operators(&,~(not),<<,>>,^(xor),`(or))
```

```
4) Logical Operators(&&,!,||)

5) Assignment Operators(=,+=,-=,*=,/=)

6) increment/decrement Operators(++,--)

7)ternary (?:)

*/



/*

const sal = 4;

const sal2 = 3;

const st = "ram";

const st1 = "dhoni";

console.log(sal + sal2);

console.log(st + " " + st1);

console.log(sal - sal2);

console.log(sal * sal2);

console.log(sal / sal2);

console.log(sal % sal2);

console.log(sal ** sal2);//4*4*4



console.log(sal == sal2);

console.log(sal === sal2);

console.log(sal != sal2);

console.log(sal > sal2);
```

```
console.log(sal < sal2);



console.log(sal > 2 && sal2 < 1);

console.log(sal > 2 || sal2 < 13);



let x = 2;

let y = 3;

console.log(x += 2);

console.log(y -= 2);

console.log(x *= 2);

console.log(y /= 2);

console.log(x += 2);

console.log(y++);

console.log(x += 2);

console.log(y -= 2);

let p = 2;

let q = 3;

console.log(++p);

console.log(--q);

console.log(p++);

console.log(q--);



const num = 22;
```

```
console.log(num % 2 == 0 ? "even" : "odd")

*/



//operator precedence



/*



//String and template literals

const firstName = 'ramu';

const lastName = "dhoni";

const job = `engineer`;

const birthYear = 2000;

const currentYear = 2024;

const sentence = "I'm " + firstName + ",a " + (currentYear - birthYear) + 'year old';

console.log(sentence);



//using template(``)

const sent = `I'm ${firstName},a ${currentYear - birthYear} year old ${job}`; //${varaible name}

console.log(sent);



//newline for strings \n\

console.log('hii \n\

good morning \n\
```

```
how r u');



*/



/*



//conditional statements



//if and else

const age = 18;

if (age >= 18) {

    console.log("Eligible to vote"); // print data in console not on the page

} else {

    console.log(" Not Eligible to vote");

}



//else if

const ag = 11;(

if (ag >= 18) {

    console.log("major")

}

else if (ag >= 12) {

    console.log("teen age")
```

```
}

else {

    console.log("kid")

}



*/



/*



//Type Conversion and coercion

//String Conversion in Concatenation

//Implicit Conversion (Type Coercion): it converts to string when it is needed or if + is with string ,value can be internally converted to string

console.log("5" + 2); // "52" (number 2 is converted to a string)

console.log("Hello " + true); // "Hello true"

//Number Conversion in Arithmetic Operations(string is converted to number internally )

console.log("5" - 2); // 3 (string "5" is converted to a number)

console.log("5" * 2); // 10

console.log("10" / "2"); // 5

// Explicit Conversion

//String Conversion:

let num = 42;

console.log(String(num)); // "42"
```

```
console.log(num.toString()); // "42"

console.log(num + ""); // "42"

//Number Conversion:

console.log(Number("42")); // 42

console.log(Number("42abc")); // NaN(not a number)

console.log(parseInt("42.5")); // 42

console.log(parseFloat("42.5")); // 42.5

//both cases example

let n = '1' + 1; //1 converted string then 11(string)

n = n - 1; // 11(string) is converted to 11(number) internally then 11-1=10

console.log(n);//10



//Truthy and False Values(Boolean Conversion:)

//In javascript 5 false values : 0, '' ,undefined ,null ,NaN ,false

//apart from falsy values remaining all are truthy values.



console.log(Boolean(0)); // false

console.log(Boolean(undefined));//flase

console.log(Boolean(NaN));//flase

console.log(Boolean(false));//flase

console.log(Boolean(''));//flase (" " or ' ' )

console.log(Boolean("Hello"));//true

console.log(Boolean({})); // true
```

```
console.log(!!0); // false

console.log(!!"world"); // true



let money = 0;

if (money) {

    console.log("money is there");

} else {

    console.log("money is not there");

}



*/



/*



//Equality Operators ==  vs ===

//1. == (Equality Operator): Performs type coercion (implicit conversion) if the types of the operands are different.Compares the values after converting them to a common type.

console.log(5 == "5");   // true (string "5" is converted to number 5)

console.log(0 == false); // true (false is converted to number 0)

console.log(null == undefined); // true (special case in JavaScript)

console.log(" " == 0);   // true (empty string is converted to number 0)

console.log("5" == true); // false ("5" is converted to number, but true is 1)

console.log([] == false); // true (empty array converts to 0)
```

```
//2.=== (Strict Equality Operator):Does not perform type coercion. Compares both value and type strictly.

console.log(5 === "5");   // false (different types)

console.log(0 === false); // false (different types)

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

console.log(" " === 0);   // false (different types)

console.log(5 === 5);//true


// Logical != (Not Equal) and Strict Not Equal !==

//Logical != (Not Equal):The != operator checks if two values are not equal with type coercion.

console.log(5 != "5");  // false (values are equal after type coercion)

console.log(5 != 6);    // true  (values are not equal)

//Strict Not Equal !== :The !== operator checks if two values are not equal without type coercion.

console.log(5 !== "5"); // true (different types, no coercion)

console.log(5 !== 5);   // false (same value and type)


//prompt :The prompt() function is used to display a dialog box that prompts the user to input some data. This is a simple way to collect user input directly from a browser. And It will always return string value.

let digit = prompt("Enter a number");

console.log(digit);


*/
```

```
/*
//Basic Boolean Logic(and, or , not)
console.log(true && true);   // true
console.log(true && false);  // false
console.log(5 > 3 && 2 < 4); // true
console.log(true || false);  // true
console.log(false || false); // false
console.log(5 > 3 || 2 > 4); // true
console.log(!true);   // false
console.log(!false);  // true
console.log(!(5 > 3)); // false

let age = 25;
let hasID = true;

if (age >= 18 && hasID) {
    console.log("You are allowed entry.");
} else {
    console.log("Access denied.");
}
*/
```

```
/*
let scoreDolphins = (96 + 108 + 89) / 3;
let scoreKoalas = (88 + 91 + 110) / 3;
if (scoreDolphins > scoreKoalas) {
    console.log("Dolphins win the trophy");
} else if (scoreKoalas > scoreDolphins) {
    console.log("Koalas win the trophy");
} else if (scoreKoalas === scoreDolphins) {
    console.log("both win the trophy");
}
*/

/*

// switch statement :A switch statement in JavaScript is used for decision-making based on multiple conditions. It provides a cleaner and more readable alternative to using multiple if-else statements when comparing a single variable or expression against multiple possible values.
let fruit = "apple";
switch (fruit) {
    case "apple":
        console.log("Apples are $1 each.");
        break;
    case "banana":
        console.log("Bananas are $0.5 each.");
```

```
        break;
    case "cherry":
        console.log("Cherries are $3 per pound.");
        break;
    default:
        console.log("Sorry, we don't have that fruit.");
}


*/


/*
//Statements and expression
let y = 5 * 2; // The expression `5 * 2` evaluates to `10`, and the result is assigned to `y`.The whole line, let y = 5 * 2;, is a statement because it performs an action: declaring a variable y and assigning a value to it.
*/


/*
//Ternary Operator(condition ? expressionIfTrue : expressionIfFalse;)
let age = 30;
let access = age >= 18 ? " Drink Allowedn🥂" : " Drink Denied😢";
console.log(access); // Output: "Allowed"
*/
```

```
/*

const bill = 275;

const tip = (bill >= 50 && bill <= 300) ? (bill * (15 / 100)) : (bill * (20 / 100));

console.log(tip);

console.log(`The bill was ${bill}, the tip was ${tip}, and the total value ${bill + tip}`);

*/



// Java Script Releases and version
```

Strict Mode in JavaScript

Strict Mode in JavaScript is a feature introduced in ECMAScript 5 that allows you to run your code in a stricter context to catch common coding errors and make your code more secure.

To enable strict mode, you can include the directive "use strict"; at the beginning of a script or function:

1) **At the script level (applies to the entire file):**

   "use strict";

   let x = 3.14; // Valid in strict mode

   y = 3.14; // ReferenceError: y is not defined

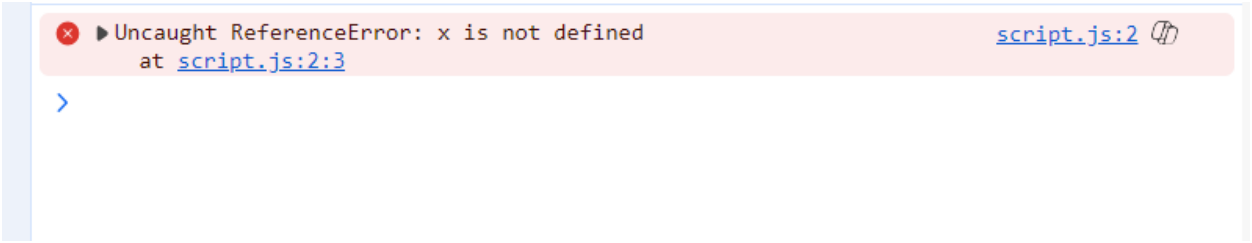
   Without Strict Mode (No Error, creates a global variable implicitly)

   x = 10; // No error, `x` becomes a global variable

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

   With Strict Mode (Throws an Error)

```
"use strict";

x = 10; // ReferenceError: x is not defined

console.log(x);
```



2) **At the function level (applies only to the specific function):**

```
function strictFunction() {
  "use strict";
  let z = 10; // Valid in strict mode
  undeclaredVar = 20; // ReferenceError: undeclaredVar is not defined
}
```

Why Use Strict Mode?

- Helps in identifying common errors early.
- Prevents the use of problematic features of JavaScript.
- Makes the code easier to debug and maintain.
- Prepare your code for future JavaScript versions.