Project – 7

JavaScript Basics

Submitted By:

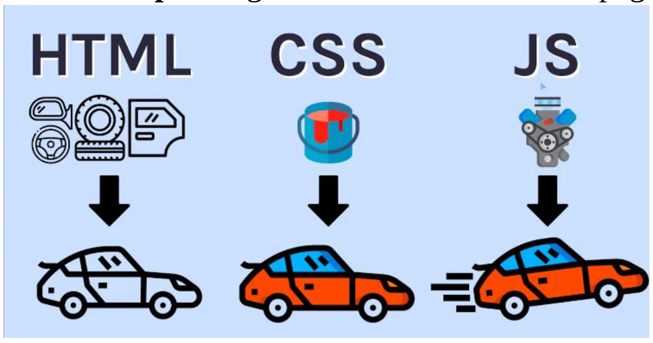
Nishigandha Patil

INDEX

INTRODUCTION TO JAVASCRIPT

➢ **What is JavaScript:**

- JavaScript is a high-level, interpreted programming language.
- Primarily used for building dynamic web pages.
- It is commonly associated with front-end web development, but it's also used on the server side (Node.js) and for various other applications.
- JavaScript is based on **ECMAScript** standard.
- **".js"** is the extension.
- JavaScript is often used in combination with HTML and CSS to create web pages that are more interactive and dynamic:
  1. **HTML:** Define the content of web pages (Structure).
  2. **CSS:** Specify the layout of web pages (Decoration).
  3. **JavaScript:** Program the behaviour of web pages (Logic).



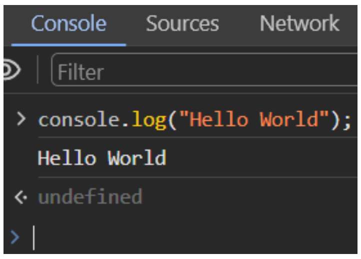
- **JS Frontend Frameworks:** React, Angular, Vue.
- **JS Backend Frameworks:** Express, Node.

➢ **JavaScript Versions:**

JavaScript was invented by 'Brendan Eich' in 1995, and became an ECMA standard in 1997. ECMAScript is the official name of the language.
Old ECMAScript versions were named by numbers: ES1, ES2, ES3, ES5, and ES6.
From 2016, versions are named by year: ES2016, 2018, 2020 ...
The 14th edition, ECMAScript 2023, was published in June 2023.

➢ **How to execute JavaScript:**

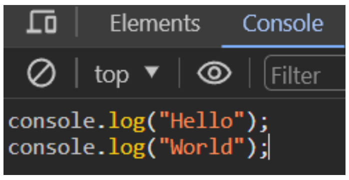
1. **Inside Browser.**
   Open Browser→ Right Click →Inspect →Console →Write JavaScript Code→Press Enter.



```
> console.log("Hello World");
  Hello World
<· undefined
> |
```

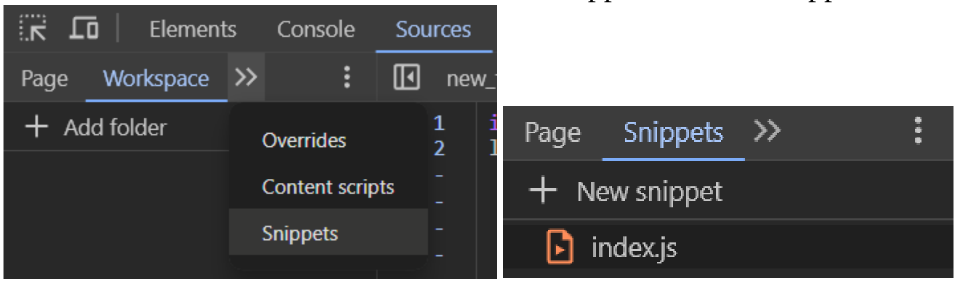
console.log () is a method used to print messages to the console, for debugging and logging information.

Press **Shift + Enter** to write multiline code in the browser.

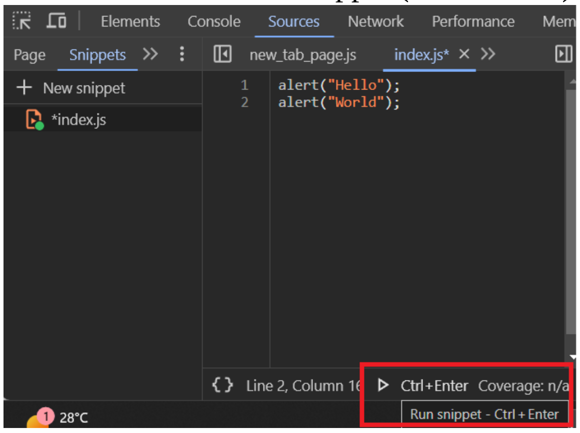


```
console.log("Hello");
console.log("World");
```

Another way to write JS code in the browser is:
Go to Sources Panel → Click on “>>” → Snippets → New snippet → Create JS file.



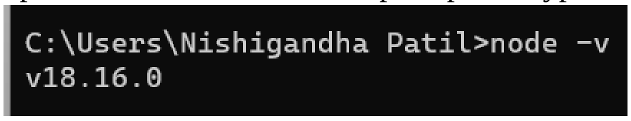
Write JS Code → Run snippet (Ctrl + Enter).



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

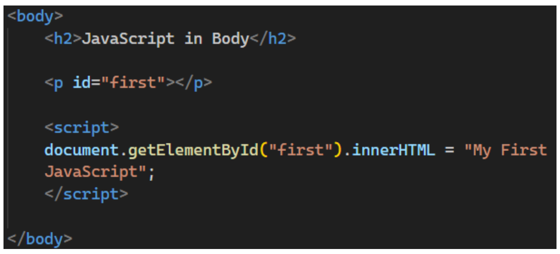
2. **Node.js**: Another way to execute JS is a runtime like Node.js which can be installed and used to run JavaScript code.
   Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It is commonly used for building server-side applications, command-line tools, and other types of scalable network programs.

   Installation Steps:
   Step1: Go to https://nodejs.org/ → Download the latest version of Node.js.
   Step2: Run the installer.
   Step3: Verify the installation: To verify that Node.js has been installed successfully, open a terminal or command prompt and type the command “node -v”:

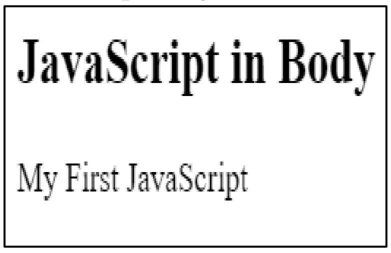


```
C:\Users\Nishigandha Patil>node -v
v18.16.0
```

3. <script> tag: You can insert JavaScript code in <script> tag of an HTML Document.



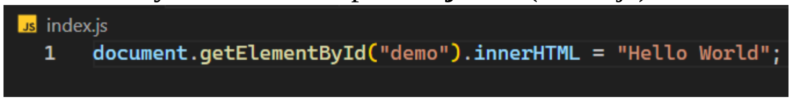
```
<body>
    <h2>JavaScript in Body</h2>

    <p id="first"></p>

    <script>
    document.getElementById("first").innerHTML = "My First
    JavaScript";
    </script>

</body>
```



JavaScript in Body

My First JavaScript

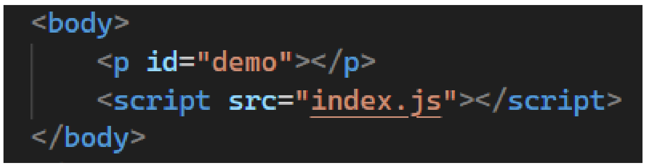
Another way is to create separate **.js** file (index.js) → Write JS Code.



index.js

```
document.getElementById("demo").innerHTML = "Hello World";
```

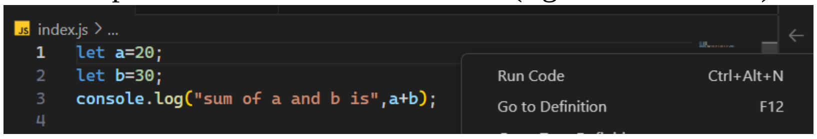
HTML file: Provide JS file name in the src attribute of script tag.



```
<body>
    <p id="demo"></p>
    <script src="index.js"></script>
</body>
```

➢ JavaScript in VS Code:

Open VS Code → Create a new file with a **.js** extension (File > New File) → Write JavaScript code and Save → Run Code (right click on code).



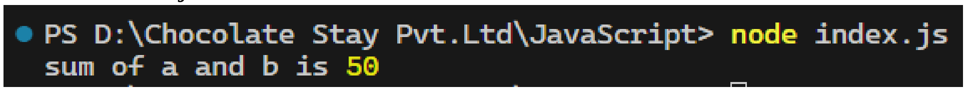
index.js > ...

```
let a=20;
let b=30;
console.log("sum of a and b is",a+b);
```

Alternatively, you can also run JavaScript code directly in the terminal. To do this, you will need to install **Node.js** and then run your JavaScript file using the "node" command, followed by the name of the file.

For example:

> node index.js

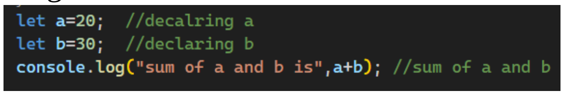


```
PS D:\Chocolate Stay Pvt.Ltd\JavaScript> node index.js
sum of a and b is 50
```

VS Code Extension for JavaScript: **JavaScript (ES6) code snippets** contains code snippets for JavaScript in ES6 (second major version of JavaScript) syntax for VS Code Editor (supports both JavaScript and TypeScript).

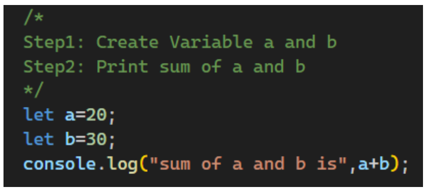
➢ Comments in JavaScript:

Single line comments start with //



```
let a=20;  //decalring a
let b=30;  //declaring b
console.log("sum of a and b is",a+b); //sum of a and b
```

Multi-line comments start with /* and end with */.



```
/*
Step1: Create Variable a and b
Step2: Print sum of a and b
*/
let a=20;
let b=30;
console.log("sum of a and b is",a+b);
```

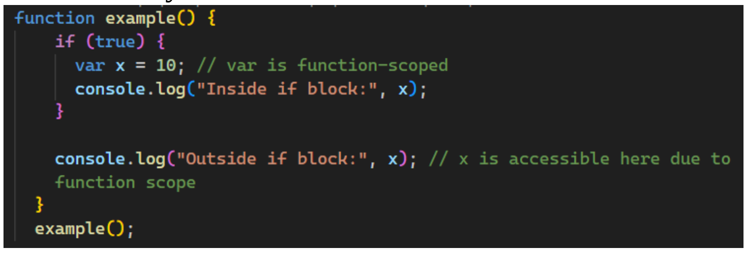
Shortcut: "ctrl + /" to add comments in VS Code.

VARIABLES

In JavaScript, variables are used to store data. They are an essential part of any programming language, as they allow you to store, retrieve, and manipulate data in your programs.

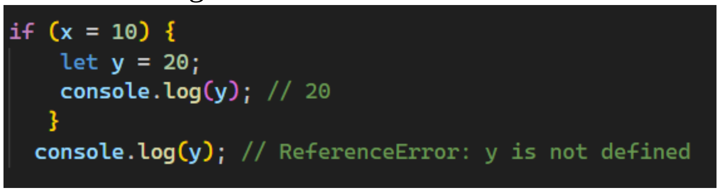
➢ **JavaScript Variables can be declared in 3 ways:**

1. **Using var:** function-scoped
   - They are only visible within the function where they are declared.
   - You can reassign values to a variable declared with var.
   - You can use the variable before it is declared in the code.
   - The var keyword was used in all JavaScript code from 1995 to 2015.
   - It should only be used in code written for older browsers.



```
function example() {
    if (true) {
      var x = 10; // var is function-scoped
      console.log("Inside if block:", x);
    }

    console.log("Outside if block:", x); // x is accessible here due to
    function scope
  }
  example();
```

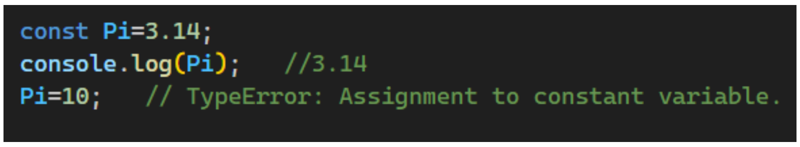
2. **Using let:** block-scoped {}
   - They are only visible within the block (a pair of curly braces) where they are declared.
   - You can reassign values to a variable declared with 'let'.



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

In this example, the "y" variable is declared with the "let" keyword and is only accessible within the block of the if statement {}. If you try to access it outside of the block, you will get a **"ReferenceError"** because "y" is not defined in that scope.

3. **Using const:** block-scoped {}
   - Cannot be reassigned once they are initialized.
   - Must be initialized with a value when they are declared and cannot be reassigned later.



```
const Pi=3.14;
console.log(Pi);   //3.14
Pi=10;   // TypeError: Assignment to constant variable.
```

In this example, the "Pi" variable is declared with the "const" keyword and is assigned the value of 3.14. If you try to reassign a new value to "Pi," you will get a **"TypeError"** because "Pi" is a constant variable and cannot be changed.

➢ var Vs let Vs const:

	var	**let**	**const**
origins	Pre ES2015	ES2015(ES6)	ES2015(ES6)
Scope	Function-scoped	Block-scoped {}	Block-scoped {}
Re-declaration	Can be redeclared	Can be redeclared	Cannot be redeclared
Re-assignment	Value can be updated	Value can be updated	Value cannot be updated

Initialization	Can or cannot be initialized
	Eg. var a;
	Output: undefined
		Can or cannot be initialized.
		Eg. let a;
		Output: undefined
			Must be initialized with a value.
			Eg., const a;
			Output: >SyntaxError
			Correct is: const a=10;

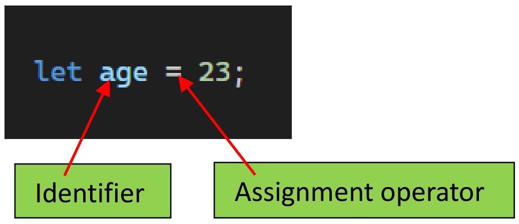
➢ Rules for constructing Variables:

- **JavaScript Identifiers:**

All JavaScript variables must be identified with unique names. These unique names are called **identifiers**. Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalArea etc).

The general rules for constructing names for variables (unique identifiers) are:

1. Names can contain letters, digits, underscores (_), and dollar ($) signs.
2. Names must begin with a letter.
3. Names can also begin with $ and _.
4. Names are case sensitive (y and Y are different variables).
5. Reserved keywords cannot be used as names.



```
let age = 23;
```

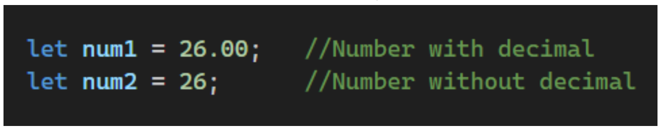
DATA TYPES

In JavaScript, there are two main types of data: primitives and objects.

➢ **Primitives:** Primitive values are immutable, meaning their values cannot be changed directly.
Simplest and most basic data types in JavaScript. They include:

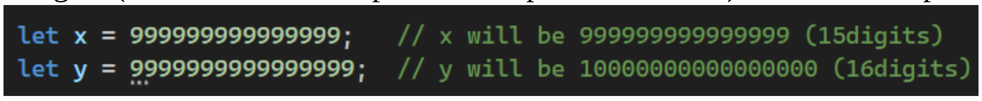
Data Types	Description
Number	Stores any Number. (eg. 10, 3.14, 1000 etc)
String	Stores any Text in Quotation. (eg. “hello”, ‘world’)
Boolean	Represents a logical value, either ‘true’ or ‘false’.
Bigint	Stores big Integer Values.
Undefined	A variable without a value. (eg., let a;)
Null	Represents the intentional absence of any object value.
Symbol	Represents a unique identifier.

1. **Number:** All JavaScript numbers are stored as decimal numbers (floating point). Numbers can be written with, or without decimals:



```
let num1 = 26.00;    //Number with decimal
let num2 = 26;       //Number without decimal
```

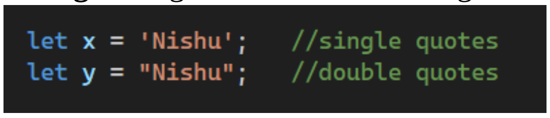
Integer Precision
Integers (numbers without a period or exponent notation) are accurate up to **15** digits.



```
let x = 999999999999999;   // x will be 999999999999999 (15digits)
let y = 9999999999999999;  // y will be 10000000000000000 (16digits)
```

The maximum number of decimals is **17**.

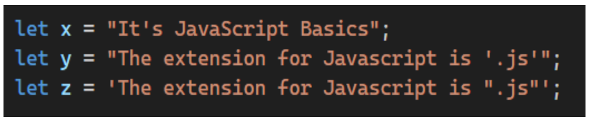
2. **String:** Strings are useful for storing text. Strings are written with quotes.



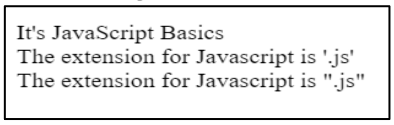
```
let x = 'Nishu';    //single quotes
let y = "Nishu";    //double quotes
```

Strings created with single or double quotes works the same.

Quotes Inside Quotes: You can use quotes inside a string.

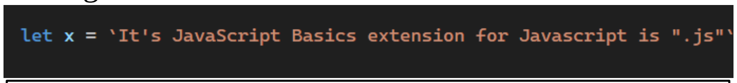


```
let x = "It's JavaScript Basics";
let y = "The extension for Javascript is '.js'";
let z = 'The extension for Javascript is ".js"';
```

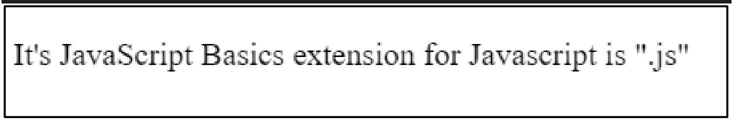


```
It's JavaScript Basics
The extension for Javascript is '.js'
The extension for Javascript is ".js"
```

Template Strings: Templates are strings enclosed in backticks (`This is a template string`). Templates allow single and double quotes, and variables(${varname} inside a string:

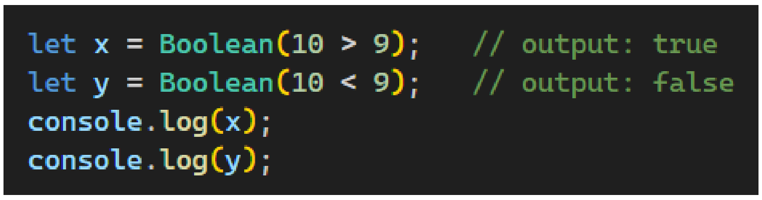


```
let x = `It's JavaScript Basics extension for Javascript is ".js"`
```



```
It's JavaScript Basics extension for Javascript is ".js"
```

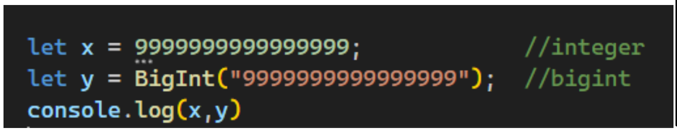
3. **Boolean:** A JavaScript Boolean represents one of two values: **true** or **false.**
You can use the **Boolean ()** function to find out if an expression (or a variable) is true:



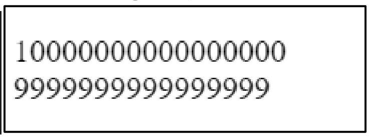
```
let x = Boolean(10 > 9);   // output: true
let y = Boolean(10 < 9);   // output: false
console.log(x);
console.log(y);
```

Everything With a "Value" is True. E.g. Boolean (10), Boolean (“false”), or any expression is true.
Everything Without a "Value" is False. The Boolean value of 0 (zero), -0 (zero), undefined, null, false is always false.

4. **Bigint:** JavaScript BigInt variables are used to store big integer values that are too big to be represented by a normal JavaScript Number.
To create a BigInt, append **n** to the end of an integer or call BigInt():



```
let x = 9999999999999999;              //integer
let y = BigInt("9999999999999999");    //bigint
console.log(x,y)
```

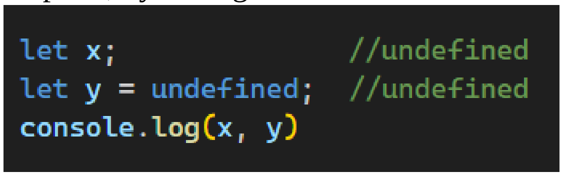


```
10000000000000000
9999999999999999
```

Notes:

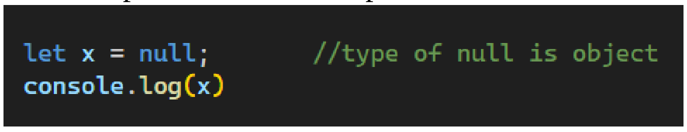
- Arithmetic between a **BigInt** and a **Number** is not allowed (type conversion lose information).
- A BigInt cannot have decimals.
- BigInt can also be written in hexadecimal, octal, or binary notation.

5. **Undefined:** A variable without a value, has the value undefined. Any variable can be emptied, by setting the value to undefined.



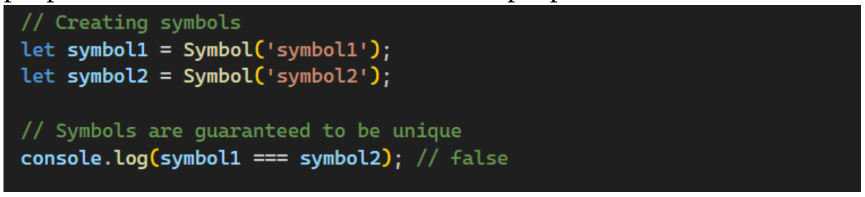
```
let x;               //undefined
let y = undefined;   //undefined
console.log(x, y)
```

6. **Null:** A special value that represents an absence of value.



```
let x = null;        //type of null is object
console.log(x)
```

7. **Symbol:** Represents a unique identifier. Symbols are often used for creating object properties that won't collide with other properties.

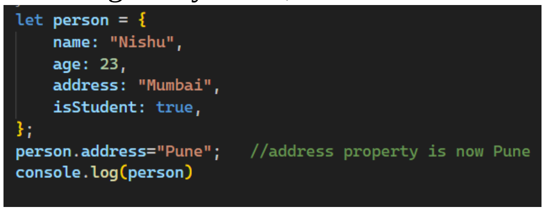


```
// Creating symbols
let symbol1 = Symbol('symbol1');
let symbol2 = Symbol('symbol2');

// Symbols are guaranteed to be unique
console.log(symbol1 === symbol2); // false
```

- **Non-primitives:** Non-primitive values are mutable, meaning their properties can be modified.

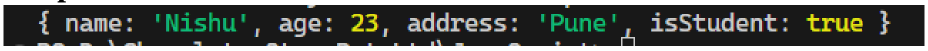
Non-Primitives	Description
Object	Represents in curly braces and has **key:value** pair.
Array	Represents an ordered list of values.
Function	A callable object that can have its own properties and be invoked.
Date	Represents a specific point in time. Date objects are static. The "clock" is not "running".

1. **Object:** Object properties are written as **key:value** pairs, separated by commas. Keys are strings or symbols, and values can be any data type, including other objects.



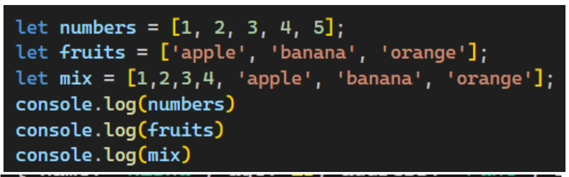
```
let person = {
    name: "Nishu",
    age: 23,
    address: "Mumbai",
    isStudent: true,
};
person.address="Pune";   //address property is now Pune
console.log(person)
```

Output:

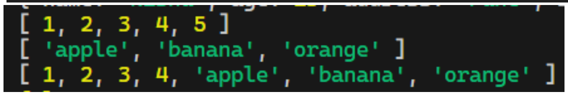


```
 { name: 'Nishu', age: 23, address: 'Pune', isStudent: true }
```

2. **An array:** An array in JavaScript is a collection of elements enclosed in **square brackets** [].

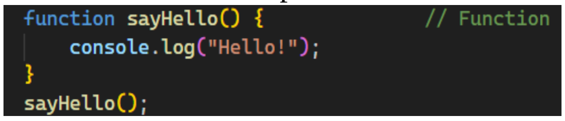


```
let numbers = [1, 2, 3, 4, 5];
let fruits = ['apple', 'banana', 'orange'];
let mix = [1,2,3,4, 'apple', 'banana', 'orange'];
console.log(numbers)
console.log(fruits)
console.log(mix)
```



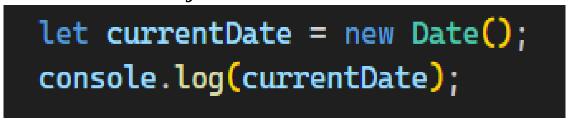
```
[ 1, 2, 3, 4, 5 ]
[ 'apple', 'banana', 'orange' ]
[ 1, 2, 3, 4, 'apple', 'banana', 'orange' ]
```

3. **Function:** A JavaScript function is defined with the function keyword.



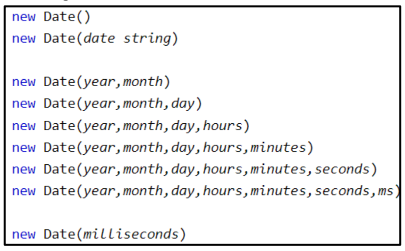
```
function sayHello() {               // Function
    console.log("Hello!");
}
sayHello();
```

4. **Date:** Date objects are created with the new Date () constructor.



```
let currentDate = new Date();
console.log(currentDate);
```

9 ways to create date:



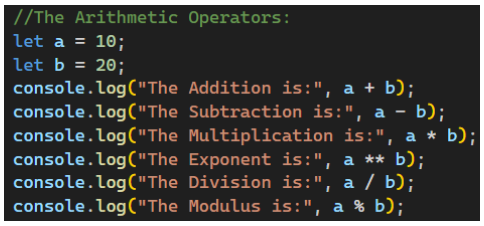
```
new Date()
new Date(date string)

new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)

new Date(milliseconds)
```

OPERATORS

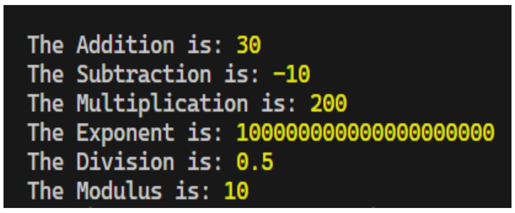
- **Operators:** JavaScript provides a variety of operators that allow you to perform different operations on values.
- **Operands:** An operand is a term used to describe the data that is manipulated by an operator. For Example: In the addition operation 3 + 5, both 3 and 5 are 'operands', and the + is the 'operator'.

1. **Arithmetic Operators:** Used to perform arithmetic on numbers.

Operators	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Return division remainder)



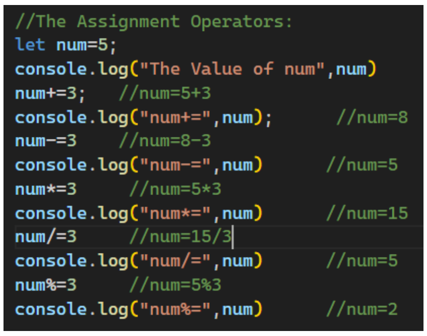
```
//The Arithmetic Operators:
let a = 10;
let b = 20;
console.log("The Addition is:", a + b);
console.log("The Subtraction is:", a - b);
console.log("The Multiplication is:", a * b);
console.log("The Exponent is:", a ** b);
console.log("The Division is:", a / b);
console.log("The Modulus is:", a % b);
```



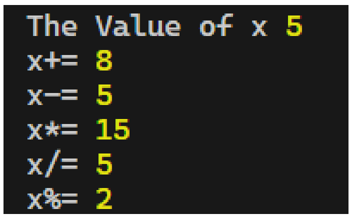
```
The Addition is: 30
The Subtraction is: -10
The Multiplication is: 200
The Exponent is: 100000000000000000000
The Division is: 0.5
The Modulus is: 10
```

2. **Assignment Operators:** Assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y



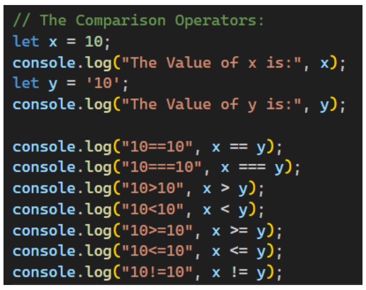
```
//The Assignment Operators:
let num=5;
console.log("The Value of num",num)
num+=3;    //num=5+3
console.log("num+=",num);       //num=8
num-=3     //num=8-3
console.log("num-=",num)       //num=5
num*=3      //num=5*3
console.log("num*=",num)       //num=15
num/=3      //num=15/3
console.log("num/=",num)       //num=5
num%=3      //num=5%3
console.log("num%=",num)       //num=2
```



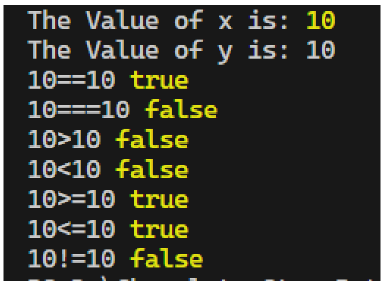
```
The Value of x 5
x+= 8
x-= 5
x*= 15
x/= 5
x%= 2
```

3. **Comparison Operators:** Used in logical statements to determine equality or difference between variables or values. It returns **true** or **false** value.

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



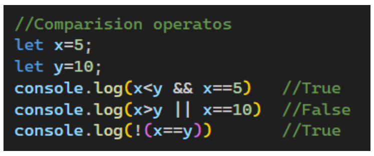
```
// The Comparison Operators:
let x = 10;
console.log("The Value of x is:", x);
let y = '10';
console.log("The Value of y is:", y);

console.log("10==10", x == y);
console.log("10===10", x === y);
console.log("10>10", x > y);
console.log("10<10", x < y);
console.log("10>=10", x >= y);
console.log("10<=10", x <= y);
console.log("10!=10", x != y);
```



```
The Value of x is: 10
The Value of y is: 10
10==10 true
10===10 false
10>10 false
10<10 false
10>=10 true
10<=10 true
10!=10 false
```

4. **Logical Operators:** Used to determine the logic between variables or values.

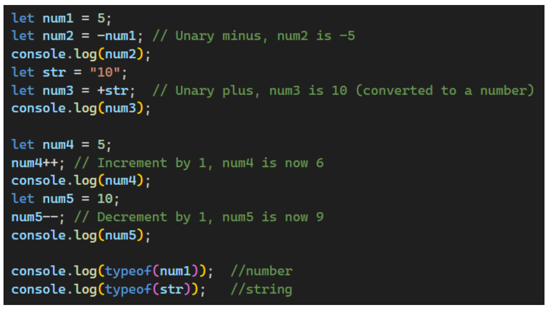
Operator	Description	Example
&&	Logical AND	(x < 10 && y > 1) is true
||	Logical OR	(x == 5 || y == 5) is false
!	Logical NOT	!(x == y) is true



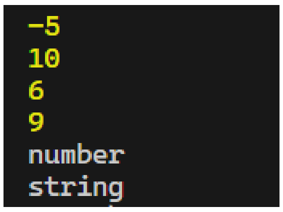
```
//Comparision operatos
let x=5;
let y=10;
console.log(x<y && x==5)    //True
console.log(x>y || x==10)   //False
console.log(!(x==y))        //True
```

5. **Unary Operators:** Works with only one operand.

Operator	Description
+	Unary plus - attempts to convert its operand to a number
-	Unary minus - negates its operand
++	Increment
--	Decrement
typeof	Returns a string representing the type of its operand



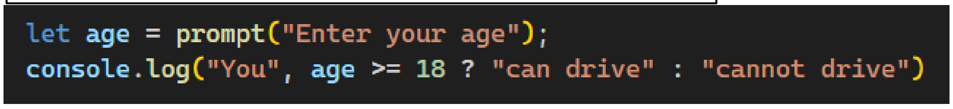
```
let num1 = 5;
let num2 = -num1; // Unary minus, num2 is -5
console.log(num2);
let str = "10";
let num3 = +str;  // Unary plus, num3 is 10 (converted to a number)
console.log(num3);

let num4 = 5;
num4++; // Increment by 1, num4 is now 6
console.log(num4);
let num5 = 10;
num5--; // Decrement by 1, num5 is now 9
console.log(num5);

console.log(typeof(num1));  //number
console.log(typeof(str));   //string
```



```
-5
10
6
9
number
string
```

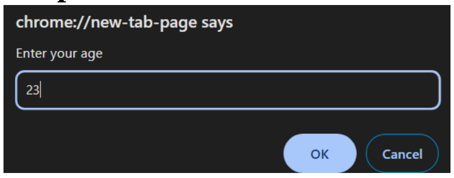
6. **Conditional (Ternary) Operator:** JavaScript contains a conditional operator that assigns a value to a variable based on some condition.
Syntax:

```
variablename = (condition) ? value1:value2
```

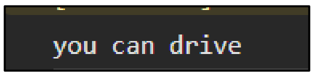


```
let age = prompt("Enter your age");
console.log("You", age >= 18 ? "can drive" : "cannot drive")
```

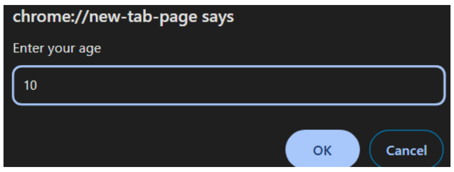
prompt () is a function used to take inputs from the user in string type.
Output:



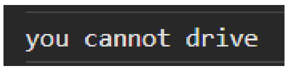
```
chrome://new-tab-page says
Enter your age
23
```



```
you can drive
```



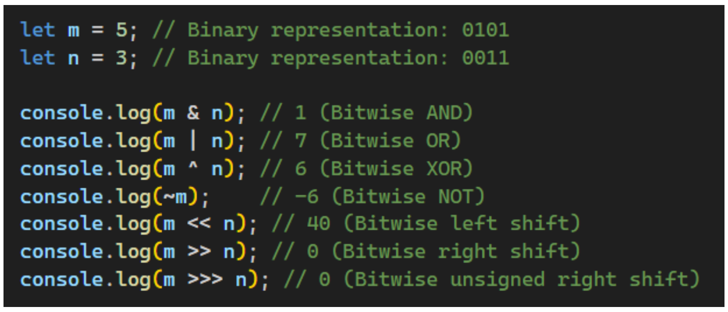
```
chrome://new-tab-page says
Enter your age
10
```



```
you cannot drive
```

7. **Bitwise Operators:** Perform operations on the binary representation of numeric values. These operators treat their operands as sequences of 32 bits (signed 32-bit integers) and perform operations bit by bit.

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1.
|	OR	Sets each bit to 1 if one of two bits is 1.
^	XOR	Sets each bit to 1 if only one of two bits is 1.
~	NOT	Inverts all the bits.
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off.
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off.
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off.

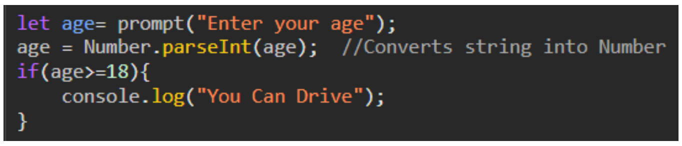


```
let m = 5; // Binary representation: 0101
let n = 3; // Binary representation: 0011

console.log(m & n); // 1 (Bitwise AND)
console.log(m | n); // 7 (Bitwise OR)
console.log(m ^ n); // 6 (Bitwise XOR)
console.log(~m);    // -6 (Bitwise NOT)
console.log(m << n); // 40 (Bitwise left shift)
console.log(m >> n); // 0 (Bitwise right shift)
console.log(m >>> n); // 0 (Bitwise unsigned right shift)
```

CONDITIONAL STATEMENTS

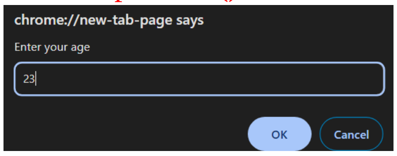
Conditional statements in JavaScript allow you to control the flow of your code based on certain conditions. The primary constructs for handling conditions are ‘if’, ‘else if’, and ‘else’.

1. **if Statement:** The if statement is used to execute a block of code if a specified condition is true.



```
let age= prompt("Enter your age");
age = Number.parseInt(age);  //Converts string into Number
if(age>=18){
    console.log("You Can Drive");
}
```

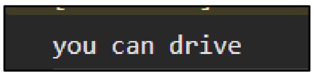
Number.parseInt(): Converts string to integer.



chrome://new-tab-page says

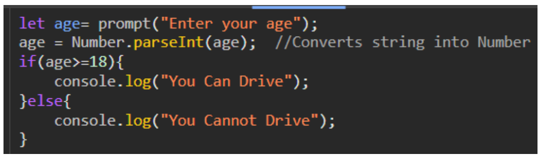
Enter your age

23



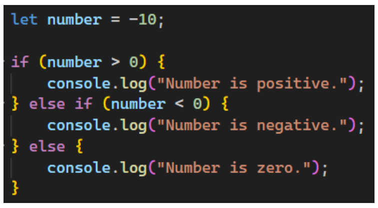
```
you can drive
```

2. **if-else Statement:** The if-else statement is used to execute one block of code if the condition is true and another block if the condition is false.

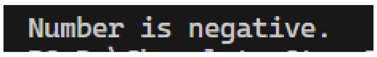


```
let age= prompt("Enter your age");
age = Number.parseInt(age);  //Converts string into Number
if(age>=18){
    console.log("You Can Drive");
}else{
    console.log("You Cannot Drive");
}
```

3. **if-else if-else Statement:** The if-else if-else statement allows you to check multiple conditions and execute different code blocks based on the first true condition.



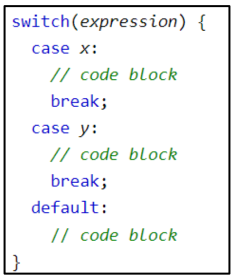
```
let number = -10;

if (number > 0) {
    console.log("Number is positive.");
} else if (number < 0) {
    console.log("Number is negative.");
} else {
    console.log("Number is zero.");
}
```



```
Number is negative.
```

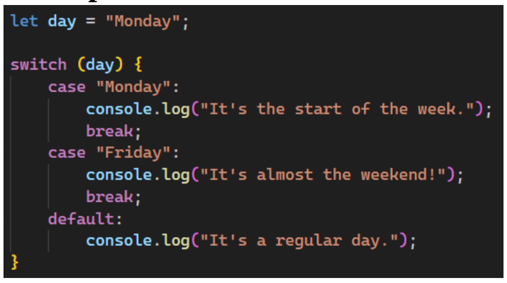
4. **Switch Statement:** Handles multiple conditions in a more concise manner.

Syntax:



```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Example:



```
let day = "Monday";

switch (day) {
    case "Monday":
        console.log("It's the start of the week.");
        break;
    case "Friday":
        console.log("It's almost the weekend!");
        break;
    default:
        console.log("It's a regular day.");
}
```

In this example, the "expression" is the variable "day" which has the value "Monday." The "expression" is compared to each of the "case" values, and when it matches the value "Monday" the corresponding block of code is executed and the message "It’s the start of the week" is printed to the console.

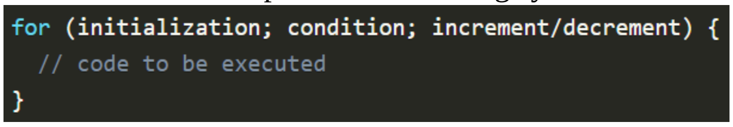
The "break" statement is used to exit the "switch" statement and prevent the code in the following cases from being executed.

LOOPS AND FUNCTIONS

JavaScript provides several types of loops to iterate over a sequence of values or execute a block of code repeatedly. Loops can execute a block of code a number of times.

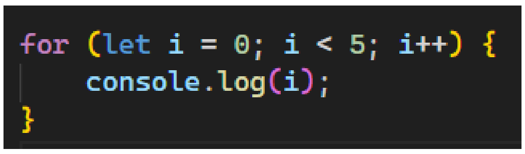
➢ **Loops:**

1. **for Loop:** Loops through a block of code a number of times.
   The standard for loop has the following syntax:

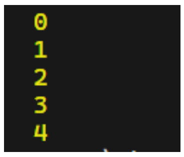


```
for (initialization; condition; increment/decrement) {
  // code to be executed
}
```

   The **initialization** the statement is executed before the loop starts and is typically used to initialize a counter variable.
   The **condition** is checked at the beginning of each iteration and if it is true, the loop continues.
   The **increment/decrement statement** is executed at the end of each iteration and is used to update the counter variable.
   Example:

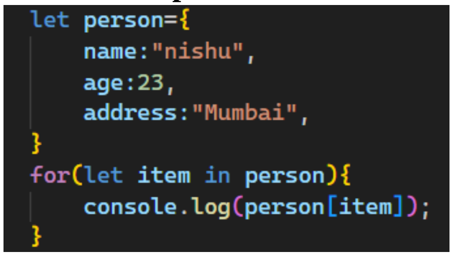


```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

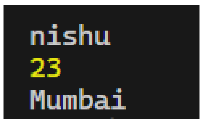


```
0
1
2
3
4
```

2. **for ... in Loop:** Used to iterate over the properties of an object.

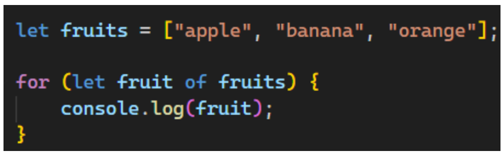


```
let person={
    name:"nishu",
    age:23,
    address:"Mumbai",
}
for(let item in person){
    console.log(person[item]);
}
```

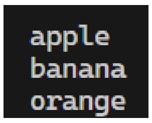


```
nishu
23
Mumbai
```

3. **for … of Loop:** Used for iterating over iterable objects, such as arrays, strings, or other iterable structures.

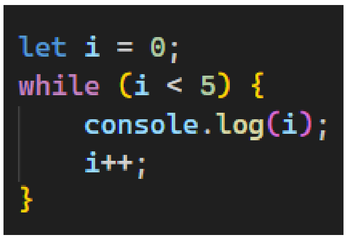


```
let fruits = ["apple", "banana", "orange"];

for (let fruit of fruits) {
    console.log(fruit);
}
```

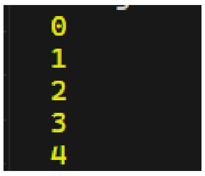


```
apple
banana
orange
```

4. **while Loop:** The while loop repeats a block of code as long as a specified condition is true.



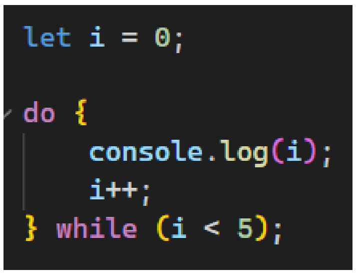
```
let i = 0;
while (i < 5) {
    console.log(i);
    i++;
}
```



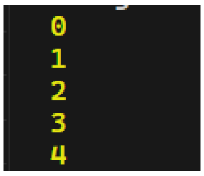
```
0
1
2
3
4
```

It's important to include a way to update the condition within the loop, otherwise it will become an infinite loop and will run forever. In the example above, the i++ statement increments the value of i by 1 at the end of each iteration, which eventually causes the condition to be false and the loop to exit.

5. **do-while Loop:** similar to the while loop, but it guarantees that the block of code is executed at least once.



```
let i = 0;

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

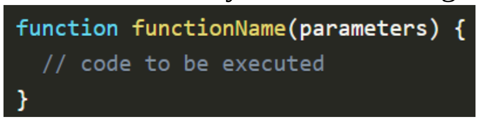


```
0
1
2
3
4
```

➢ **Functions:**

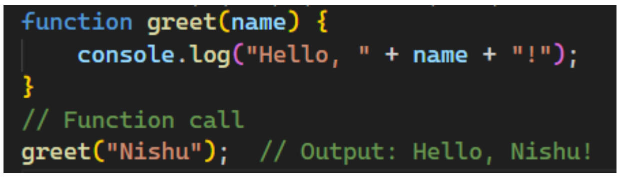
Functions in JavaScript are blocks of reusable code that can be defined and called to perform a specific task.

Here's the basic syntax for defining a function in JavaScript:



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

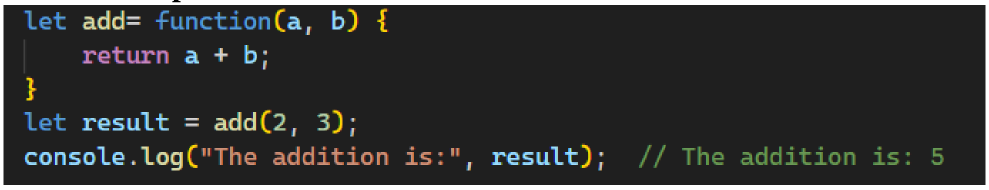
1. **Function Declaration:**



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

In this example, **greet** is a function that takes a parameter **name** and logs a greeting to the console. The function is then called with the argument "Nishu".

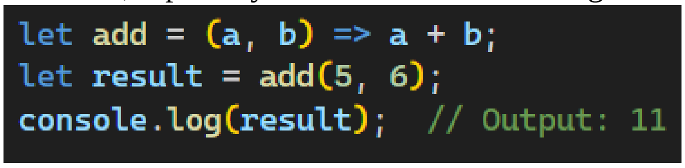
2. **Function Expression:**



```
let add= function(a, b) {
    return a + b;
}
let result = add(2, 3);
console.log("The addition is:", result);  // The addition is: 5
```

In this case, the add function takes two parameters, a and b, and returns their sum.

3. **Arrow Function:** Arrow functions provide a concise syntax for defining functions, especially useful for short and straightforward functions.

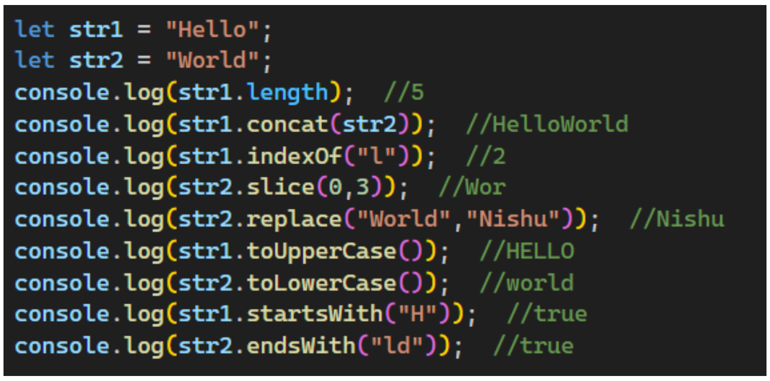


```
let add = (a, b) => a + b;
let result = add(5, 6);
console.log(result);  // Output: 11
```

STRING METHODS

JavaScript provides a number of built-in methods for manipulating strings. Some of the most commonly used string methods are:

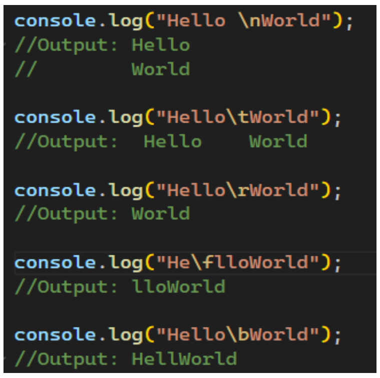
1. **length:** This method returns the number of characters in a string.
2. **concat:** This method is used to concatenate (combine) two or more strings.
3. **indexOf:** This method is used to find the index of a specific character or substring in a string.
4. **slice (start, [end]):** This method is used to extract a portion of a string.
5. **replace (search, replacement):** This method is used to replace a specific character or substring in a string.
6. **toUpperCase and toLowerCase:** These methods are used to convert a string to uppercase or lowercase letters.
7. **startsWith(prefix) / endsWith(suffix):** Checks if a string starts/ends with a specified substring.



```javascript
let str1 = "Hello";
let str2 = "World";
console.log(str1.length);  //5
console.log(str1.concat(str2));  //HelloWorld
console.log(str1.indexOf("l"));  //2
console.log(str2.slice(0,3));  //Wor
console.log(str2.replace("World","Nishu"));  //Nishu
console.log(str1.toUpperCase());  //HELLO
console.log(str2.toLowerCase());  //world
console.log(str1.startsWith("H"));  //true
console.log(str2.endsWith("ld"));  //true
```

➢ **Escape Characters:**

1. **Backslash:** turns special characters into string characters.
   - \” Inserts a double quote in a string.
   - \’ Inserts a single quote in a string
   - \\ inserts backslash
2. **\n:** new line
3. **\t:** tab
4. **\r:** carriage return
5. **\f:** form feed
6. **\b:** backspace

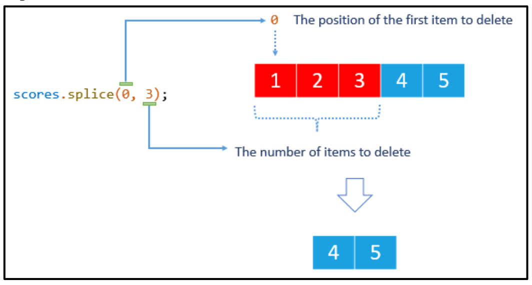


```javascript
console.log("Hello \nWorld");
//Output: Hello
//        World

console.log("Hello\tWorld");
//Output:  Hello    World

console.log("Hello\rWorld");
//Output: World

console.log("He\flloWorld");
//Output: lloWorld

console.log("Hello\bWorld");
//Output: HellWorld
```

ARRAY METHODS

JavaScript provides a number of built-in methods for manipulating arrays. Some of the most commonly used array methods are:

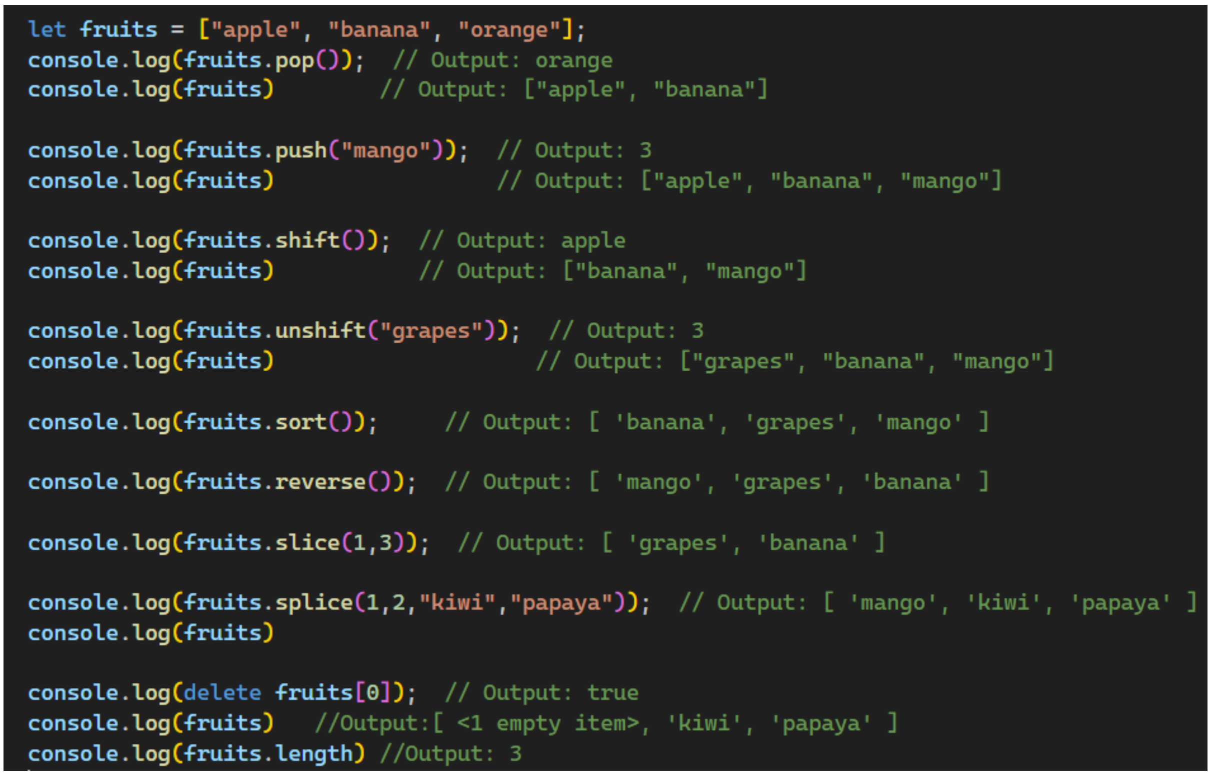
1. **pop:** Removes the last element from an array and returns that element.
2. **push:** Adds one or more elements to the end of an array. Return the length of an array.
3. **shift:** Removes the first element from an array and returns that element.
4. **unshift:** Adds one or more elements to the beginning of an array.
5. **sort:** Used to sort the elements of an array.
6. **reverse:** Used to reverse the order of elements in an array.
7. **slice:** Returns a shallow copy of a portion of an array.
8. **splice:** Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. Overwrites the original array.
   Syntax:



```
scores.splice(0, 3);
```

0
The position of the first item to delete
1 2 3 4 5
The number of items to delete
4 5

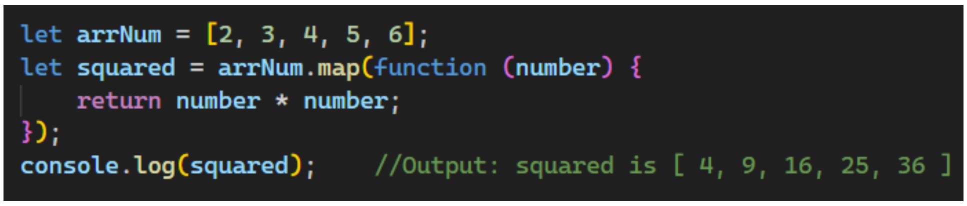
9. **delete:** Array element can be deleted using delete function. Length of array remains as it is.



```
let fruits = ["apple", "banana", "orange"];
console.log(fruits.pop());  // Output: orange
console.log(fruits)         // Output: ["apple", "banana"]

console.log(fruits.push("mango"));  // Output: 3
console.log(fruits)                 // Output: ["apple", "banana", "mango"]

console.log(fruits.shift());  // Output: apple
console.log(fruits)           // Output: ["banana", "mango"]

console.log(fruits.unshift("grapes"));  // Output: 3
console.log(fruits)                    // Output: ["grapes", "banana", "mango"]

console.log(fruits.sort());     // Output: [ 'banana', 'grapes', 'mango' ]

console.log(fruits.reverse());  // Output: [ 'mango', 'grapes', 'banana' ]

console.log(fruits.slice(1,3));  // Output: [ 'grapes', 'banana' ]

console.log(fruits.splice(1,2,"kiwi","papaya"));  // Output: [ 'mango', 'kiwi', 'papaya' ]
console.log(fruits)

console.log(delete fruits[0]);  // Output: true
console.log(fruits)   //Output:[ <1 empty item>, 'kiwi', 'papaya' ]
console.log(fruits.length) //Output: 3
```

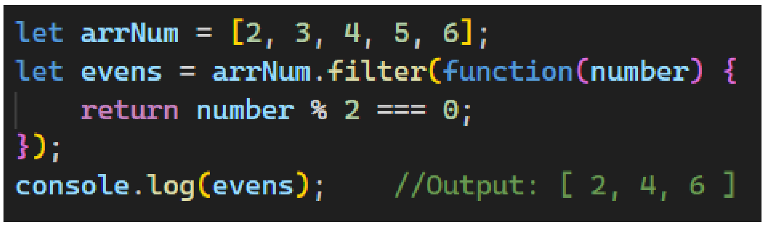
➢ Higher Order Array Methods:

1. **map ():** Applies a function to each element in the array and returns a new array of the results.



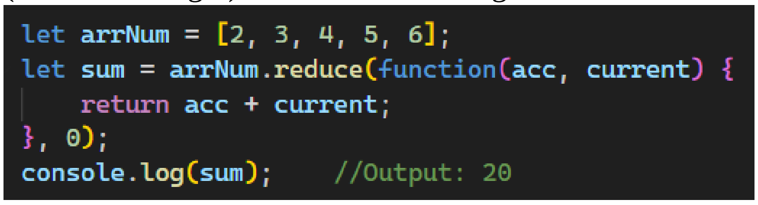
```
let arrNum = [2, 3, 4, 5, 6];
let squared = arrNum.map(function (number) {
    return number * number;
});
console.log(squared);    //Output: squared is [ 4, 9, 16, 25, 36 ]
```

2. **filter ():** Creates a new array containing only the elements that satisfy a provided condition.



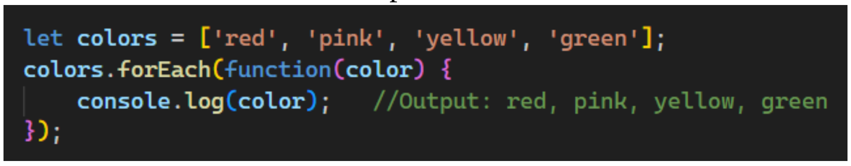
```
let arrNum = [2, 3, 4, 5, 6];
let evens = arrNum.filter(function(number) {
    return number % 2 === 0;
});
console.log(evens);    //Output: [ 2, 4, 6 ]
```

3. **reduce ():** Applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.



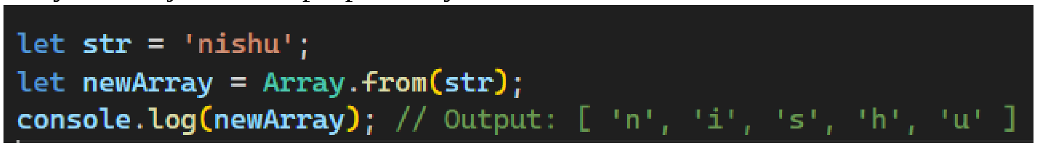
```
let arrNum = [2, 3, 4, 5, 6];
let sum = arrNum.reduce(function(acc, current) {
    return acc + current;
}, 0);
console.log(sum);    //Output: 20
```

4. **forEach method:** Executes a provided function once for each array element.



```
let colors = ['red', 'pink', 'yellow', 'green'];
colors.forEach(function(color) {
    console.log(color);   //Output: red, pink, yellow, green
});
```

5. **Array.from:** Static method in JavaScript that creates a new, shallow-copied Array instance from an array-like or iterable object. It allows you to convert an iterable or array-like object into a proper array.



```
let str = 'nishu';
let newArray = Array.from(str);
console.log(newArray); // Output: [ 'n', 'i', 's', 'h', 'u' ]
```

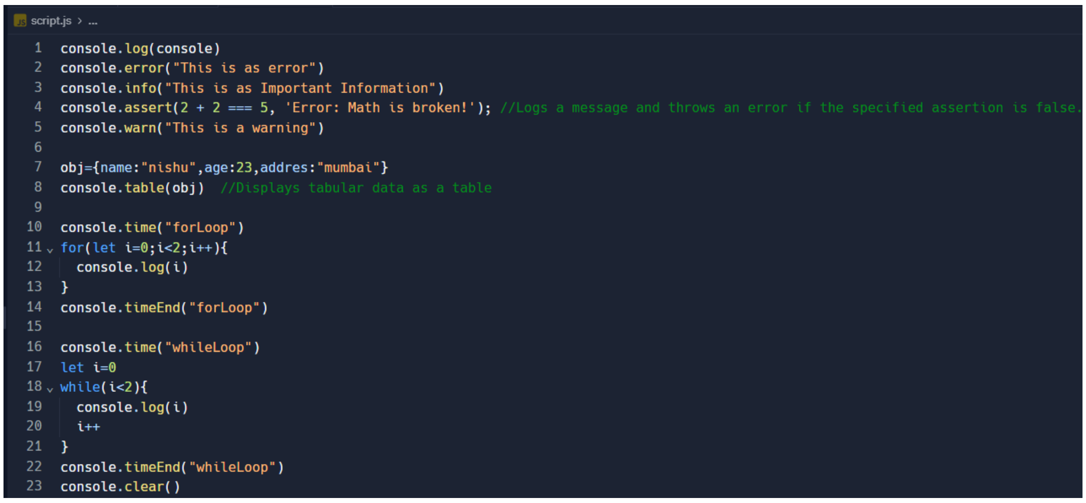
JAVASCRIPT IN BROWSER

Browser has an embedded engine called the JavaScript Engine or the JavaScript Runtime.

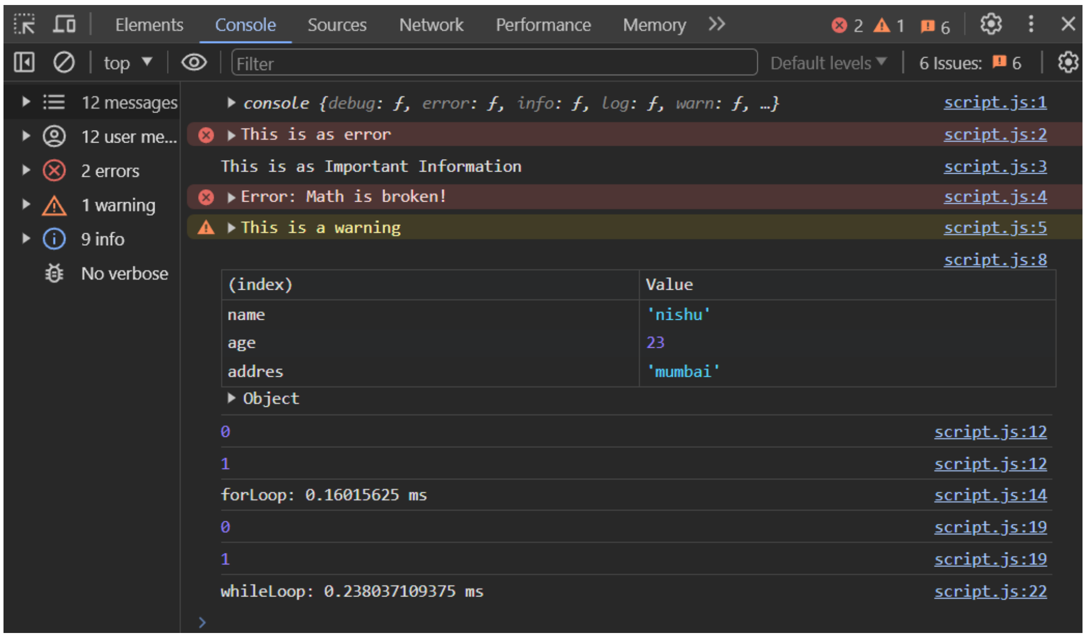
Open Browser→ Right Click →Inspect →Console →Write JavaScript Code→Press Enter.

➢ **Console Object Methods:** Provides a set of methods for logging messages and interacting with the browser's console

1. **console.log ():** Outputs information to the console.
2. **console.error ():** Outputs an error message to the console with an error icon .
3. **console.info ():** Outputs a warning message to the console with an info icon.
4. **console.assert ():** Used for logging an error message to the console if the specified expression evaluates to false.
5. **console.warn ():** Outputs a warning message to the console with a warning icon.
6. **console.table ():** Displays tabular data as a table.
7. **console.time () and console.timeEnd ():** Measures the time taken for a block of code to execute.
8. **console.clear():** Clears the console.



```
console.log(console)
console.error("This is as error")
console.info("This is as Important Information")
console.assert(2 + 2 === 5, 'Error: Math is broken!'); //Logs a message and throws an error if the specified assertion is false.
console.warn("This is a warning")

obj={name:"nishu",age:23,addres:"mumbai"}
console.table(obj)  //Displays tabular data as a table

console.time("forLoop")
for(let i=0;i<2;i++){
  console.log(i)
}
console.timeEnd("forLoop")

console.time("whileLoop")
let i=0
while(i<2){
  console.log(i)
  i++
}
console.timeEnd("whileLoop")
console.clear()
```



```
▸ console {debug: ƒ, error: ƒ, info: ƒ, log: ƒ, warn: ƒ, …}     script.js:1
▸ This is as error                                              script.js:2
This is as Important Information                                script.js:3
▸ Error: Math is broken!                                        script.js:4
▸ This is a warning                                             script.js:5
                                                                script.js:8
```

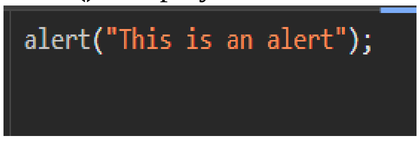
(index)	Value
name	'nishu'
age	23
addres	'mumbai'

```
▸ Object
0                                                               script.js:12
1                                                               script.js:12
forLoop: 0.16015625 ms                                          script.js:14
0                                                               script.js:19
1                                                               script.js:19
whileLoop: 0.238037109375 ms                                    script.js:22
>
```

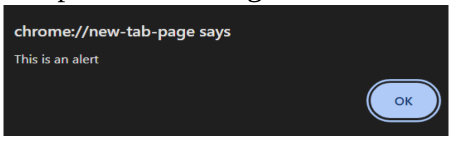
➢ Alert, Prompt, Confirm Methods:

In JavaScript, **alert**, **prompt**, and **confirm** are methods provided by the window object that allow you to interact with the user through simple dialog boxes.

1. **alert ():** Displays an alert box with a specified message and an OK button.



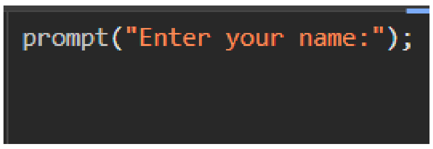
```
alert("This is an alert");
```



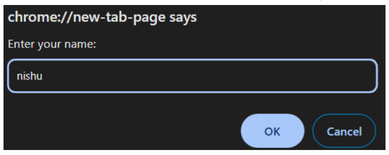
chrome://new-tab-page says

This is an alert

2. **prompt ():** Displays a dialog box that prompts the user for input. The prompt function returns the user's input as a string. If the user clicks "Cancel," it returns null.



```
prompt("Enter your name:");
```

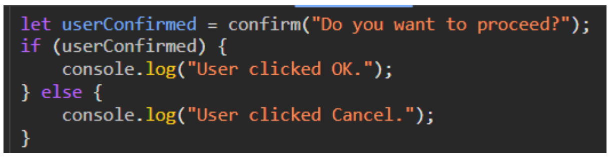


chrome://new-tab-page says

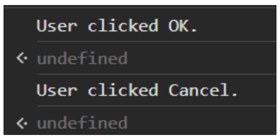
Enter your name:

nishu

3. **confirm ():** Displays a dialog box with a specified message and OK/Cancel buttons. It returns true if the user clicks "OK" and false if the user clicks "Cancel."



```
let userConfirmed = confirm("Do you want to proceed?");
if (userConfirmed) {
    console.log("User clicked OK.");
} else {
    console.log("User clicked Cancel.");
}
```



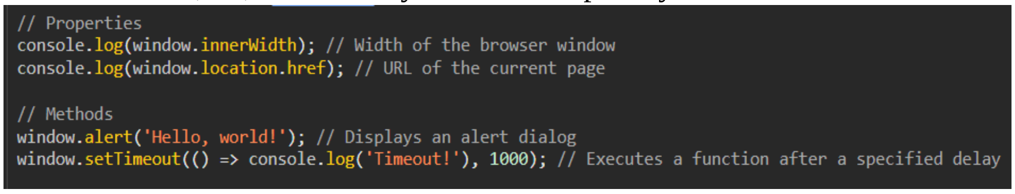
```
  User clicked OK.
<· undefined
  User clicked Cancel.
<· undefined
```

➢ Window Object, DOM, BOM:

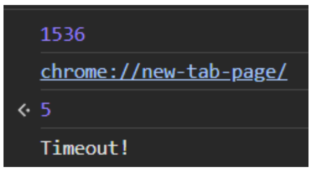
In web development using JavaScript, the window object, the Document Object Model (DOM), and the Browser Object Model (BOM) are three crucial concepts that play key roles in creating dynamic and interactive web pages.

1. Window Object:

The window object is the global object in a browser's JavaScript environment.
It represents the browser window or a tab and provides various properties and methods.
It serves as the global scope for JavaScript, meaning variables and functions declared without the var, let, or const keywords are implicitly defined on the window object.

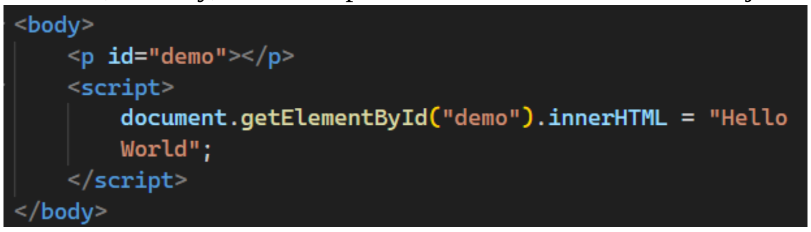


```
// Properties
console.log(window.innerWidth); // Width of the browser window
console.log(window.location.href); // URL of the current page

// Methods
window.alert('Hello, world!'); // Displays an alert dialog
window.setTimeout(() => console.log('Timeout!'), 1000); // Executes a function after a specified delay
```



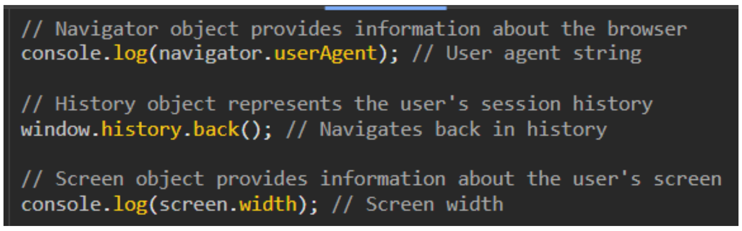
```
  1536
  chrome://new-tab-page/
<· 5
  Timeout!
```

2. **Document Object Model (DOM):**
   The DOM is a programming interface provided by browsers, representing the structure of an HTML or XML document as a tree of objects.
   It allows JavaScript to interact with and manipulate the content, structure, and style of a document.
   Elements in an HTML document become nodes in the DOM tree, and JavaScript can traverse, modify, and manipulate these nodes. window Object:

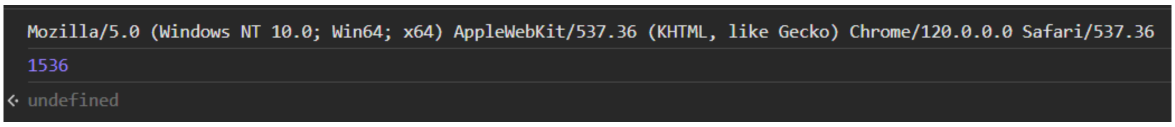


```
<body>
    <p id="demo"></p>
    <script>
        document.getElementById("demo").innerHTML = "Hello
        World";
    </script>
</body>
```

3. **Browser Object Model (BOM):**
   The BOM represents additional objects provided by the browser, which are not directly related to the document structure. It includes objects like navigator, history, and screen.



```
// Navigator object provides information about the browser
console.log(navigator.userAgent); // User agent string

// History object represents the user's session history
window.history.back(); // Navigates back in history

// Screen object provides information about the user's screen
console.log(screen.width); // Screen width
```



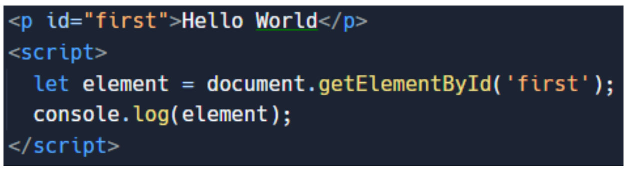
```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36
1536
< undefined
```

WORKING WITH DOM

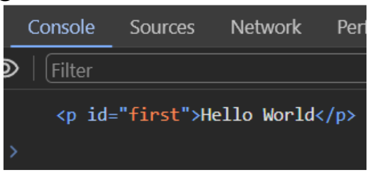
JavaScript allows you to dynamically manipulate the content and structure of a web page. Here are some common operations you can perform with the DOM:

➢ **Selecting Elements:**

1. **getElementById ():** Used to get an element with given “id” attribute.

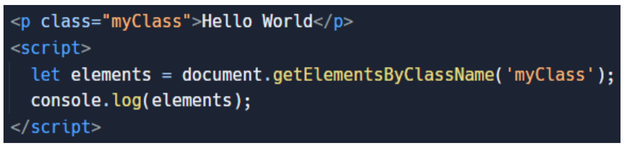


```
<p id="first">Hello World</p>
<script>
  let element = document.getElementById('first');
  console.log(element);
</script>
```

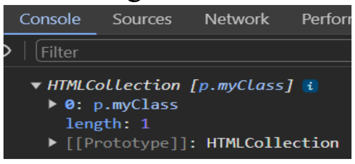


```
<p id="first">Hello World</p>
```

2. **getElementsByClassName ():** Used to get an element with given “class” attribute.

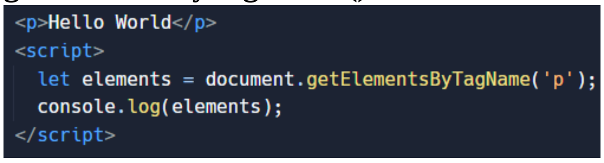


```
<p class="myClass">Hello World</p>
<script>
  let elements = document.getElementsByClassName('myClass');
  console.log(elements);
</script>
```

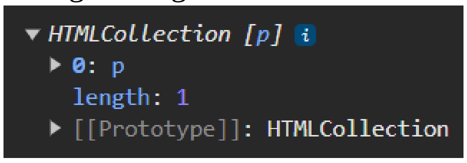


```
▼ HTMLCollection [p.myClass] i
  ▶ 0: p.myClass
    length: 1
  ▶ [[Prototype]]: HTMLCollection
```

3. **getElementsByTagName ():** Returns elements with given tag name.

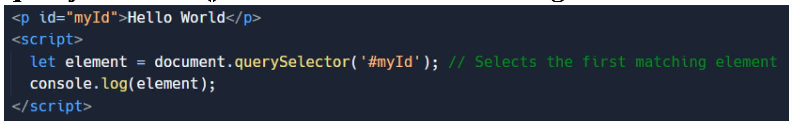


```
<p>Hello World</p>
<script>
  let elements = document.getElementsByTagName('p');
  console.log(elements);
</script>
```

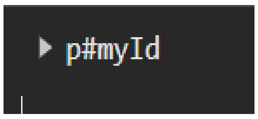


```
▼ HTMLCollection [p] i
  ▶ 0: p
    length: 1
  ▶ [[Prototype]]: HTMLCollection
```

4. **querySelector ():** Selects the first matching element.

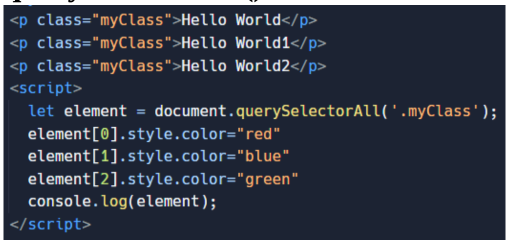


```
<p id="myId">Hello World</p>
<script>
  let element = document.querySelector('#myId'); // Selects the first matching element
  console.log(element);
</script>
```

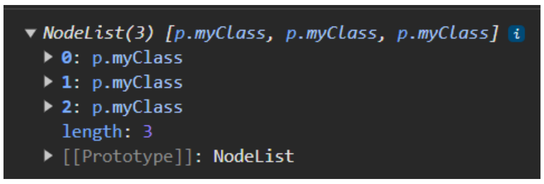


```
▶ p#myId
```

5. **querySelectorAll ():** Selects all matching elements.



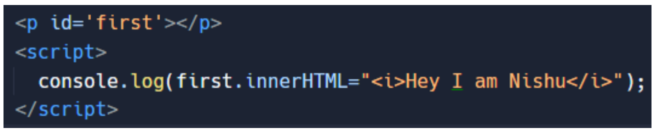
```
<p class="myClass">Hello World</p>
<p class="myClass">Hello World1</p>
<p class="myClass">Hello World2</p>
<script>
  let element = document.querySelectorAll('.myClass');
  element[0].style.color="red"
  element[1].style.color="blue"
  element[2].style.color="green"
  console.log(element);
</script>
```



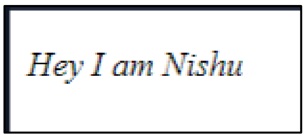
```
▼ NodeList(3) [p.myClass, p.myClass, p.myClass] i
  ▶ 0: p.myClass
  ▶ 1: p.myClass
  ▶ 2: p.myClass
    length: 3
  ▶ [[Prototype]]: NodeList
```

➢ **Modifying Elements:**

1. **innerHTML:** Allows you to get or set the HTML content within an element.

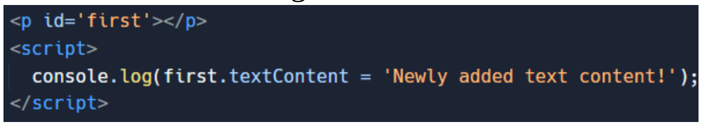


```
<p id='first'></p>
<script>
  console.log(first.innerHTML="<i>Hey I am Nishu</i>");
</script>
```

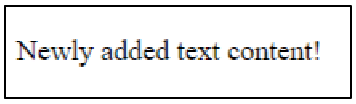


Hey I am Nishu

2. **textContent:** Used to get or set the text content of an element.

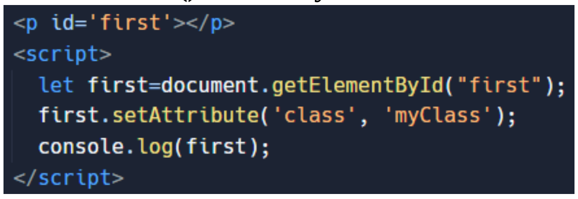


```
<p id='first'></p>
<script>
  console.log(first.textContent = 'Newly added text content!');
</script>
```

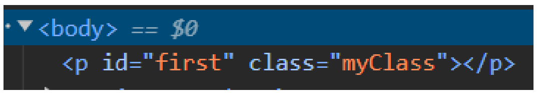


Newly added text content!

3. **setAttribute ()**: Allows you to set the value of an attribute on an HTML element.

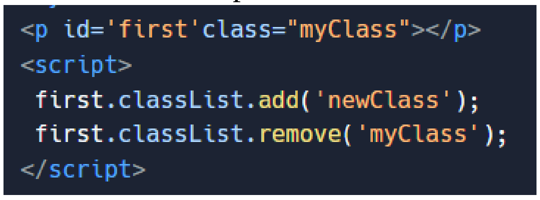


```
<p id='first'></p>
<script>
  let first=document.getElementById("first");
  first.setAttribute('class', 'myClass');
  console.log(first);
</script>
```

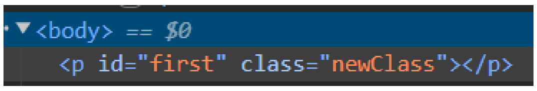


```
▼<body> == $0
   <p id="first" class="myClass"></p>
```

4. **classList ()**It is a read-only property that returns a live DOMTokenList collection representing the classes of an element. It provides methods to add, remove, toggle, and check for the presence of CSS classes on an HTML element.



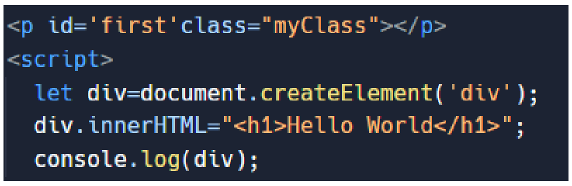
```
<p id='first'class="myClass"></p>
<script>
 first.classList.add('newClass');
 first.classList.remove('myClass');
</script>
```



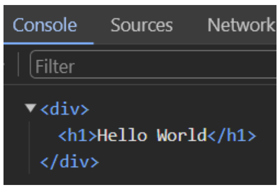
```
▼<body> == $0
   <p id="first" class="newClass"></p>
```

➢ **Creating and Appending Elements:**

1. **createElement ():** Used to create new HTML elements.

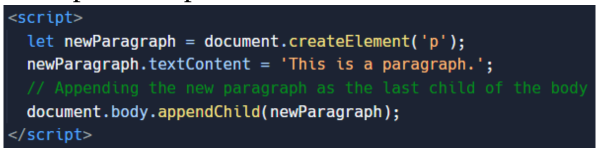


```
<p id='first'class="myClass"></p>
<script>
  let div=document.createElement('div');
  div.innerHTML="<h1>Hello World</h1>";
  console.log(div);
```

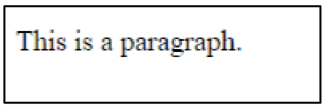


```
▼<div>
   <h1>Hello World</h1>
 </div>
```

2. **appendChild ():** Used to append a node (an element, text node, etc.) as the last child of a specified parent node.



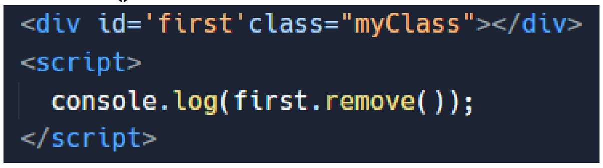
```
<script>
  let newParagraph = document.createElement('p');
  newParagraph.textContent = 'This is a paragraph.';
  // Appending the new paragraph as the last child of the body
  document.body.appendChild(newParagraph);
</script>
```



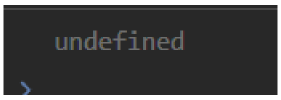
This is a paragraph.

➢ **Removing Elements:**

1. **remove ():** Used to remove the element.



```
<div id='first'class="myClass"></div>
<script>
  console.log(first.remove());
</script>
```



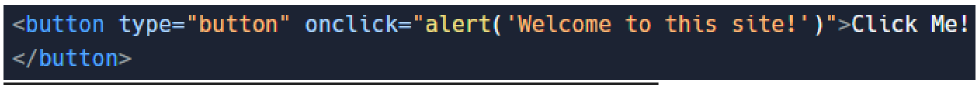
```
  undefined
>
```

BROWSER EVENTS

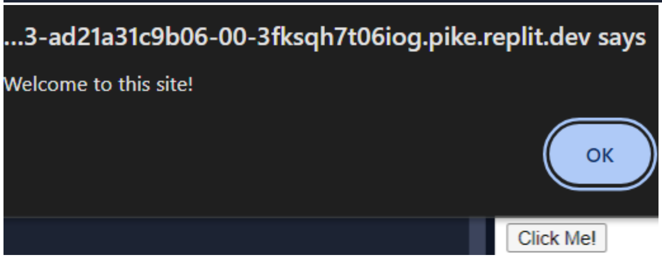
Events in JavaScript are interactions or occurrences that happen in the browser, such as user actions or changes in the browser environment. JavaScript allows you to respond to these events and execute specific code when they occur.

- Some important DOM events are:
  1. **Mouse Events:** 'click', 'mouseover', 'mouseout', 'mousemove', etc.
  2. **Keyboard Events:** 'keydown', 'keyup', 'keypress'.
  3. **Form Events:** 'submit', 'change', 'input', etc.
  4. **Focus Events:** 'focus', 'blur', 'focusin', 'focusout'.
  5. **Window Events:** 'load', 'resize', 'unload', etc.

- **Handling Events:** Events can be handled through HTML attributes.
  - **Event Listeners:** Event listeners are functions that listen for a specific type of event on a particular DOM element. They are attached using the **addEventListener** method.



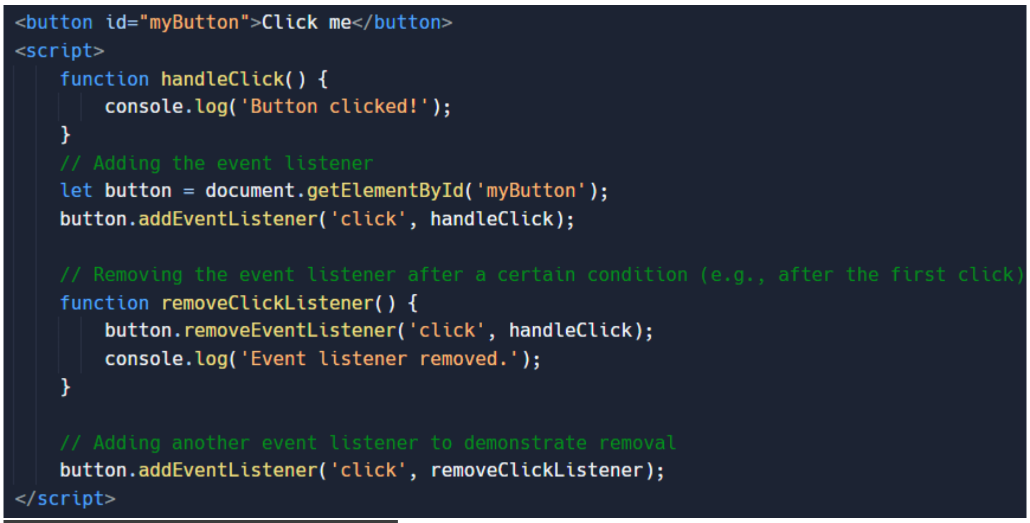
```
<button type="button" onclick="alert('Welcome to this site!')">Click Me!
</button>
```



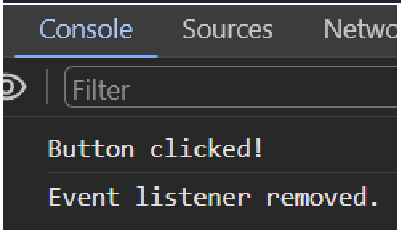
...3-ad21a31c9b06-00-3fksqh7t06iog.pike.replit.dev says

Welcome to this site!

  - **addEventListener** and **removeEventListener:**



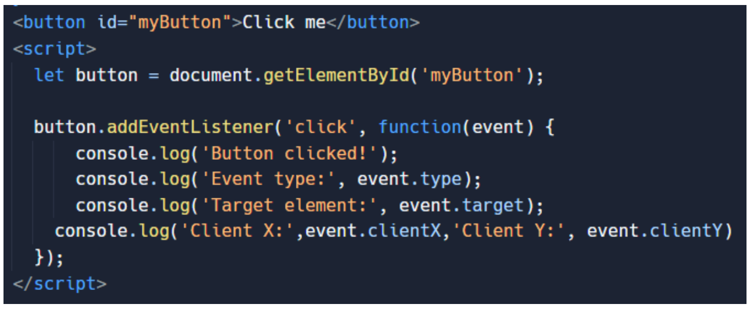
```
<button id="myButton">Click me</button>
<script>
    function handleClick() {
        console.log('Button clicked!');
    }
    // Adding the event listener
    let button = document.getElementById('myButton');
    button.addEventListener('click', handleClick);

    // Removing the event listener after a certain condition (e.g., after the first click)
    function removeClickListener() {
        button.removeEventListener('click', handleClick);
        console.log('Event listener removed.');
    }

    // Adding another event listener to demonstrate removal
    button.addEventListener('click', removeClickListener);
</script>
```



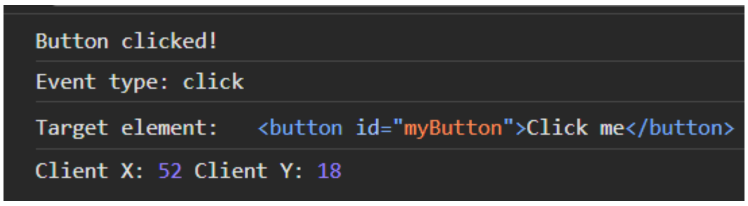
```
Button clicked!
Event listener removed.
```

removeEventListener works if and only if the function object is same.

- **Event Object:** When an event occurs, an event object is created and passed to the event handler. It contains information about the event, such as the type of event, the target element, and more.



```
<button id="myButton">Click me</button>
<script>
  let button = document.getElementById('myButton');

  button.addEventListener('click', function(event) {
      console.log('Button clicked!');
      console.log('Event type:', event.type);
      console.log('Target element:', event.target);
    console.log('Client X:',event.clientX,'Client Y:', event.clientY)
  });
</script>
```

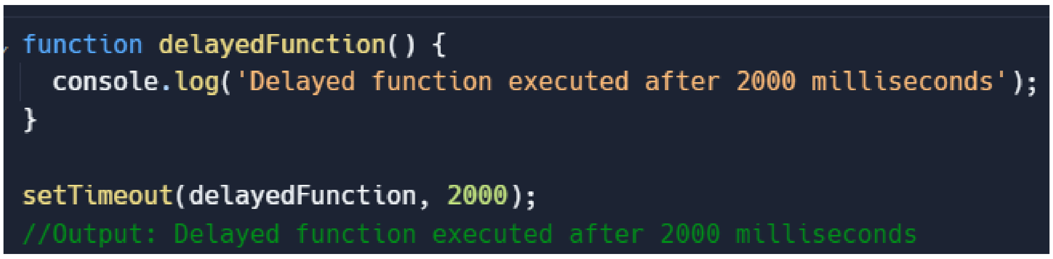


```
Button clicked!
Event type: click
Target element:   <button id="myButton">Click me</button>
Client X: 52 Client Y: 18
```

Here event is an event object. It returns pointer events in console.

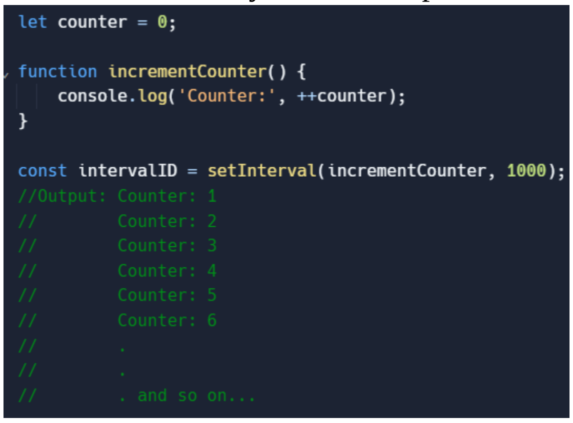
- **setTimeout and setInterval:**
These are two functions in JavaScript that allow you to execute code after a specified amount of time. They are often used for asynchronous operations, animations, and periodic tasks.

1. **setTimeout:** Used to execute a function or a piece of code after a specified delay (in milliseconds). It allows you to schedule a function to run once after a certain amount of time.



```
function delayedFunction() {
  console.log('Delayed function executed after 2000 milliseconds');
}

setTimeout(delayedFunction, 2000);
//Output: Delayed function executed after 2000 milliseconds
```

2. **setInterval:** Used to repeatedly execute a function or a piece of code at fixed intervals. It allows you to create periodic tasks.



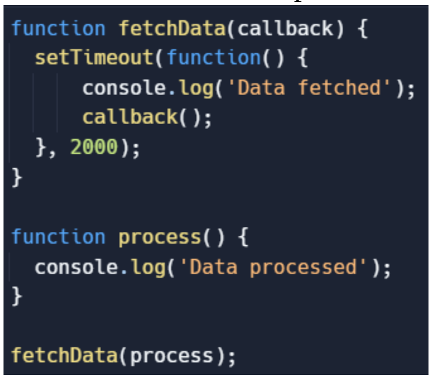
```
let counter = 0;

function incrementCounter() {
    console.log('Counter:', ++counter);
}

const intervalID = setInterval(incrementCounter, 1000);
//Output: Counter: 1
//        Counter: 2
//        Counter: 3
//        Counter: 4
//        Counter: 5
//        Counter: 6
//        .
//        .
//        . and so on...
```

ASYNCHRONOUS PROGRAMMING

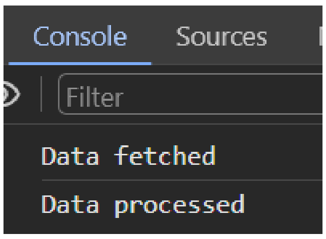
Asynchronous programming in JavaScript is a paradigm that allows you to execute code non-blocking, meaning that certain operations can be initiated, and the program can continue to run other tasks without waiting for those operations to complete. This is especially important in web development, where tasks such as fetching data from a server or handling user interactions are inherently asynchronous.

There are several mechanisms in JavaScript for handling asynchronous programming:

1. **Callbacks:**
   Callbacks are functions that are passed as arguments to other functions and are executed after the completion of an asynchronous operation.



```
function fetchData(callback) {
  setTimeout(function() {
      console.log('Data fetched');
      callback();
  }, 2000);
}

function process() {
  console.log('Data processed');
}

fetchData(process);
```



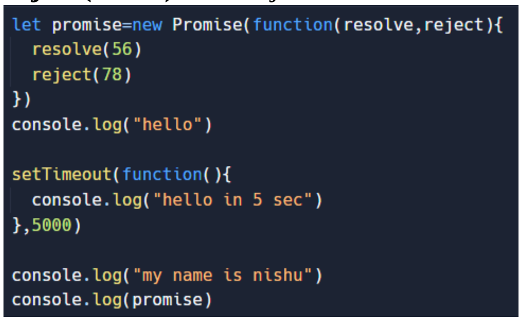
```
Data fetched
Data processed
```

   But the problem of callback function is the pyramid of doom/callback hell; as calls become more nested the code becomes deeper and more difficult to manage. Solution to callback hell is Promises.

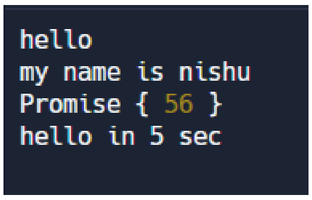
2. **Promise:**
   Promises provide a more structured way to handle asynchronous operations. They represent a value that may be available now, in the future, or never. The code either executes or fails, in both cases the subscriber will be notified.
   Syntax:

   let promise = new promise (function (resolve, reject) {
   //executer
   });

   resolve(value): If the job is finished successfully.
   reject(value): If the job fails.

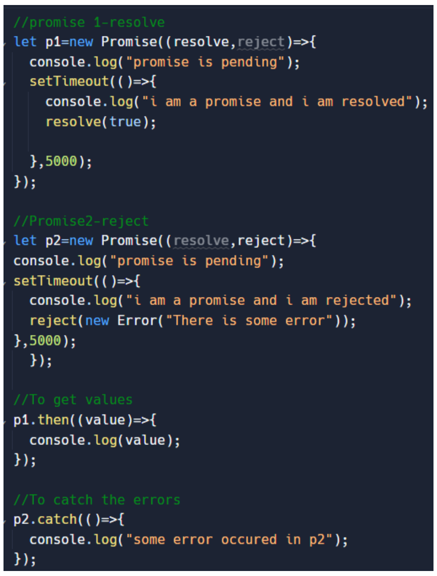


```
let promise=new Promise(function(resolve,reject){
  resolve(56)
  reject(78)
})
console.log("hello")

setTimeout(function(){
  console.log("hello in 5 sec")
},5000)

console.log("my name is nishu")
console.log(promise)
```

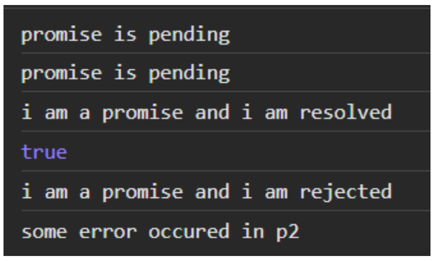


```
hello
my name is nishu
Promise { 56 }
hello in 5 sec
```

- **then and catch methods:**
  The **then** method is used to attach callbacks that will be called when the Promise is fulfilled, meaning that the asynchronous operation has completed successfully. The **catch** method is used to handle errors that occur during the asynchronous operation or in any of the preceding then handlers.

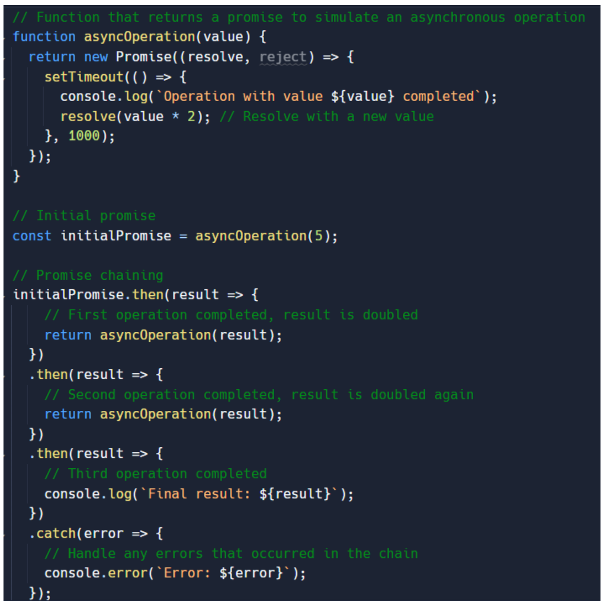


```javascript
//promise 1-resolve
let p1=new Promise((resolve,reject)=>{
  console.log("promise is pending");
  setTimeout(()=>{
    console.log("i am a promise and i am resolved");
    resolve(true);

  },5000);
});

//Promise2-reject
let p2=new Promise((resolve,reject)=>{
console.log("promise is pending");
setTimeout(()=>{
  console.log("i am a promise and i am rejected");
  reject(new Error("There is some error"));
},5000);
  });

//To get values
p1.then((value)=>{
  console.log(value);
});

//To catch the errors
p2.catch(()=>{
  console.log("some error occured in p2");
});
```

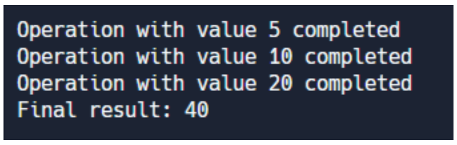


```
promise is pending
promise is pending
i am a promise and i am resolved
true
i am a promise and i am rejected
some error occured in p2
```

- **Promise chaining:** It is a powerful feature in JavaScript that allows you to execute a sequence of asynchronous operations one after another. Each operation returns a promise, and you can chain additional operations using the .then() method.

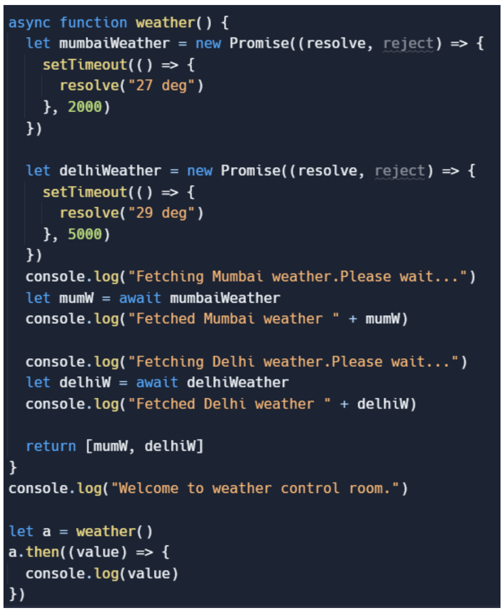


```javascript
// Function that returns a promise to simulate an asynchronous operation
function asyncOperation(value) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(`Operation with value ${value} completed`);
      resolve(value * 2); // Resolve with a new value
    }, 1000);
  });
}

// Initial promise
const initialPromise = asyncOperation(5);

// Promise chaining
initialPromise.then(result => {
    // First operation completed, result is doubled
    return asyncOperation(result);
  })
  .then(result => {
    // Second operation completed, result is doubled again
    return asyncOperation(result);
  })
  .then(result => {
    // Third operation completed
    console.log(`Final result: ${result}`);
  })
  .catch(error => {
    // Handle any errors that occurred in the chain
    console.error(`Error: ${error}`);
  });
```

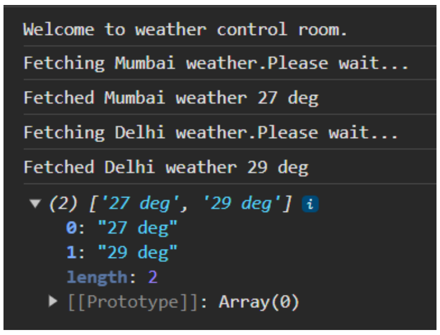


```
Operation with value 5 completed
Operation with value 10 completed
Operation with value 20 completed
Final result: 40
```

3. **async/await:** Modern and more readable way to work with asynchronous code. It is built on top of promises and allows writing asynchronous code that looks and behaves like synchronous code.
   The await function works only inside async function. It makes JS wait until the promise settle returns its value.

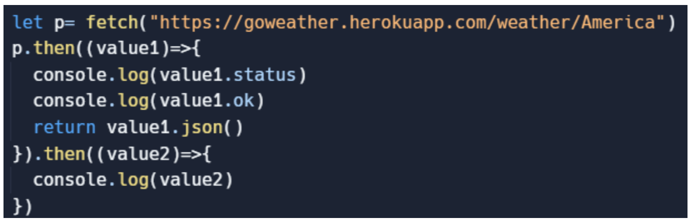


```
async function weather() {
  let mumbaiWeather = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("27 deg")
    }, 2000)
  })

  let delhiWeather = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("29 deg")
    }, 5000)
  })
  console.log("Fetching Mumbai weather.Please wait...")
  let mumW = await mumbaiWeather
  console.log("Fetched Mumbai weather " + mumW)

  console.log("Fetching Delhi weather.Please wait...")
  let delhiW = await delhiWeather
  console.log("Fetched Delhi weather " + delhiW)

  return [mumW, delhiW]
}
console.log("Welcome to weather control room.")

let a = weather()
a.then((value) => {
  console.log(value)
})
```

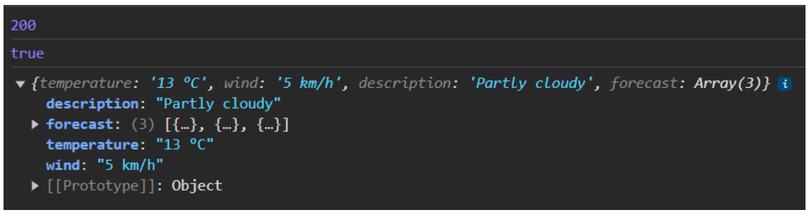


```
Welcome to weather control room.
Fetching Mumbai weather.Please wait...
Fetched Mumbai weather 27 deg
Fetching Delhi weather.Please wait...
Fetched Delhi weather 29 deg
▼ (2) ['27 deg', '29 deg']
    0: "27 deg"
    1: "29 deg"
    length: 2
  ▶ [[Prototype]]: Array(0)
```

4. **Fetch API:** Used for making asynchronous HTTP requests to fetch data from a server.



```
let p= fetch("https://goweather.herokuapp.com/weather/America")
p.then((value1)=>{
  console.log(value1.status)
  console.log(value1.ok)
  return value1.json()
}).then((value2)=>{
  console.log(value2)
})
```

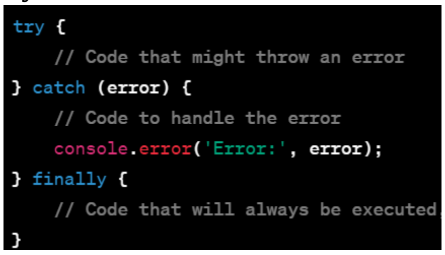


```
200
true
▼ {temperature: '13 °C', wind: '5 km/h', description: 'Partly cloudy', forecast: Array(3)}
    description: "Partly cloudy"
  ▶ forecast: (3) [{…}, {…}, {…}]
    temperature: "13 °C"
    wind: "5 km/h"
  ▶ [[Prototype]]: Object
```

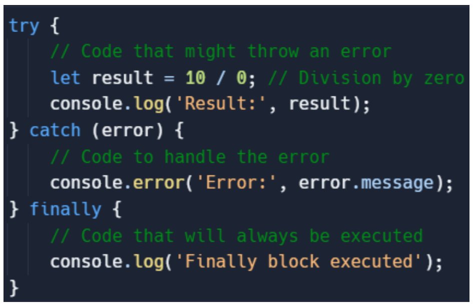
ERROR HANDLING

Error handling in JavaScript is crucial for writing robust and reliable code. There are several mechanisms for handling errors, including try-catch blocks, throwing custom errors.

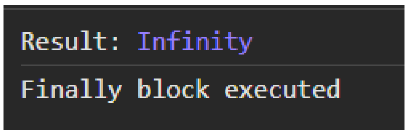
1. **Try-Catch Blocks:**
   The try statement allows you to define a block of code to be tested for errors while the catch statement allows you to handle the error.
   Syntax:



```
try {
    // Code that might throw an error
} catch (error) {
    // Code to handle the error
    console.error('Error:', error);
} finally {
    // Code that will always be executed
}
```



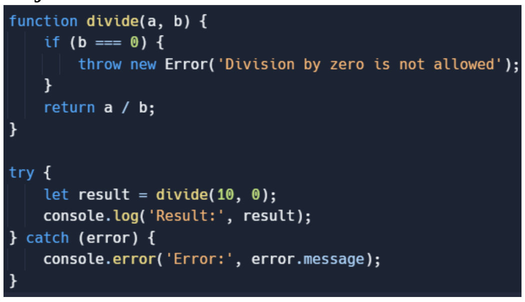
```
try {
    // Code that might throw an error
    let result = 10 / 0; // Division by zero
    console.log('Result:', result);
} catch (error) {
    // Code to handle the error
    console.error('Error:', error.message);
} finally {
    // Code that will always be executed
    console.log('Finally block executed');
}
```



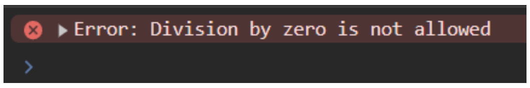
```
Result: Infinity
Finally block executed
```

The **'finally'** block is optional, and if present, it is guaranteed to be executed.

2. **Throwing Custom Errors:**
   You can throw custom errors using the throw statement, creating instances of the Error object or its subclasses.

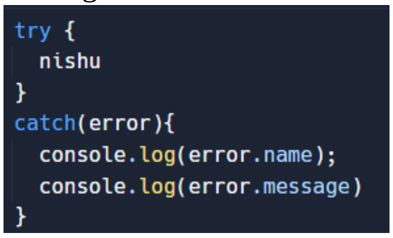


```
function divide(a, b) {
    if (b === 0) {
        throw new Error('Division by zero is not allowed');
    }
    return a / b;
}

try {
    let result = divide(10, 0);
    console.log('Result:', result);
} catch (error) {
    console.error('Error:', error.message);
}
```

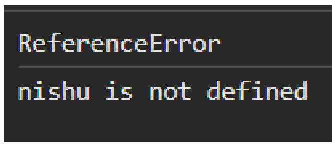


```
▸ Error: Division by zero is not allowed
>
```

Error Object: For all built-in errors the error object has two main properties: Name and message.



```
try {
  nishu
}
catch(error){
  console.log(error.name);
  console.log(error.message)
}
```

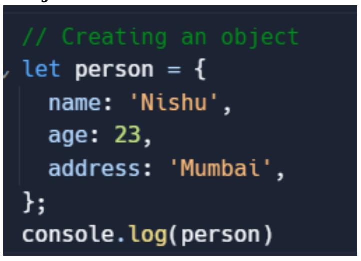


```
ReferenceError
nishu is not defined
```

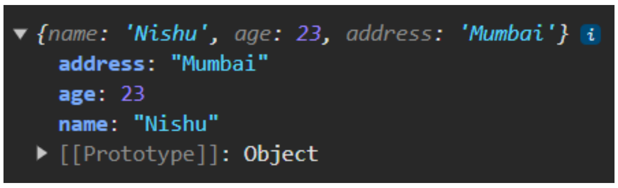
OBJECT ORIENTED PROGRAMMING

JavaScript supports object-oriented programming (OOP) principles, allowing you to create and work with objects, encapsulation, inheritance, and polymorphism.

1. **Objects:** In JavaScript, everything is an object or can be treated as an object. Objects are instances of classes, which are essentially templates for creating objects.

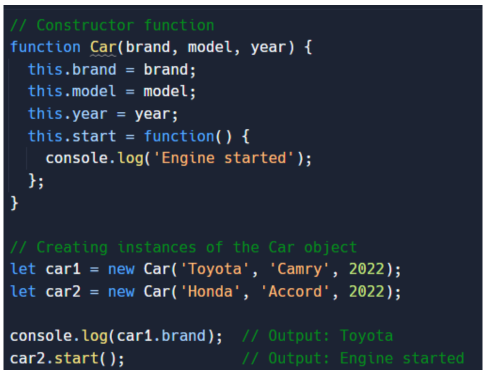


```
// Creating an object
let person = {
  name: 'Nishu',
  age: 23,
  address: 'Mumbai',
};
console.log(person)
```



```
▼ {name: 'Nishu', age: 23, address: 'Mumbai'} i
    address: "Mumbai"
    age: 23
    name: "Nishu"
  ▶ [[Prototype]]: Object
```

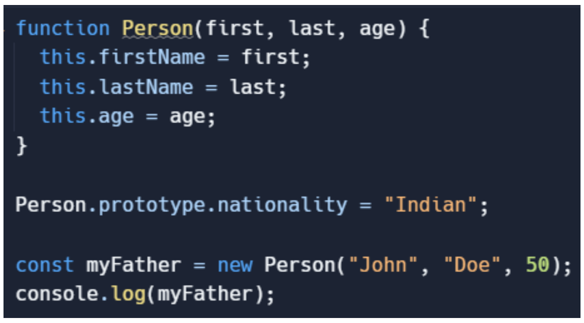
2. **Constructor functions**: Used to create objects with a common set of properties and methods.



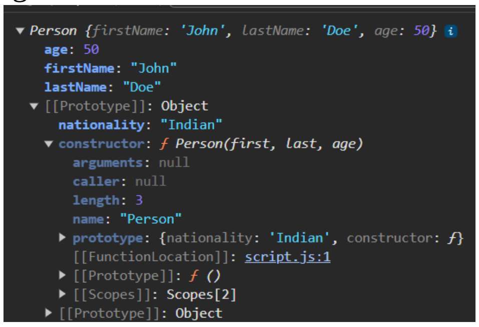
```
// Constructor function
function Car(brand, model, year) {
  this.brand = brand;
  this.model = model;
  this.year = year;
  this.start = function() {
    console.log('Engine started');
  };
}

// Creating instances of the Car object
let car1 = new Car('Toyota', 'Camry', 2022);
let car2 = new Car('Honda', 'Accord', 2022);

console.log(car1.brand);  // Output: Toyota
car2.start();             // Output: Engine started
```

this keyword refers to an object.

3. **Prototype**: Prototypes allow you to share properties and methods among all instances of a constructor function, reducing memory usage.



```
function Person(first, last, age) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
}

Person.prototype.nationality = "Indian";

const myFather = new Person("John", "Doe", 50);
console.log(myFather);
```

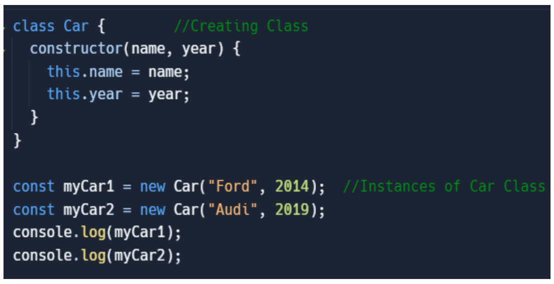


```
▼ Person {firstName: 'John', lastName: 'Doe', age: 50} i
    age: 50
    firstName: "John"
    lastName: "Doe"
  ▼ [[Prototype]]: Object
      nationality: "Indian"
    ▼ constructor: ƒ Person(first, last, age)
        arguments: null
        caller: null
        length: 3
        name: "Person"
      ▶ prototype: {nationality: 'Indian', constructor: ƒ}
        [[FunctionLocation]]: script.js:1
      ▶ [[Prototype]]: ƒ ()
      ▶ [[Scopes]]: Scopes[2]
    ▶ [[Prototype]]: Object
```

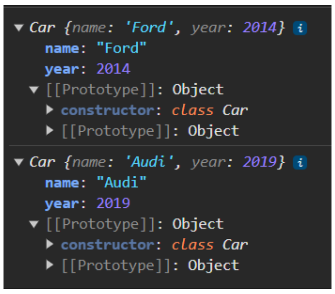
4. **Classes:** Classes are template for creating an object. Use the keyword class to create a class. Always add a method named constructor ():

Syntax:

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



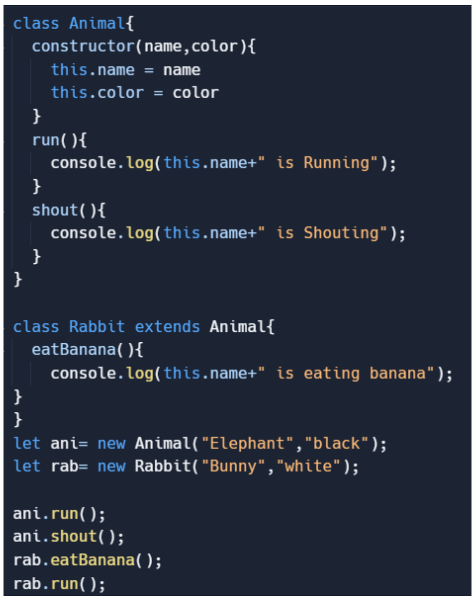
```
class Car {        //Creating Class
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}

const myCar1 = new Car("Ford", 2014);  //Instances of Car Class
const myCar2 = new Car("Audi", 2019);
console.log(myCar1);
console.log(myCar2);
```



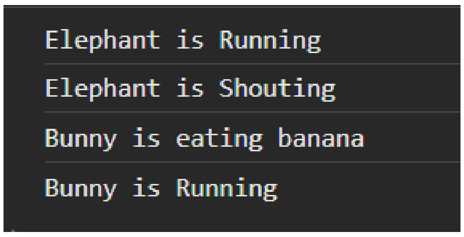
```
▼ Car {name: 'Ford', year: 2014} ℹ
    name: "Ford"
    year: 2014
  ▼ [[Prototype]]: Object
    ▶ constructor: class Car
    ▶ [[Prototype]]: Object
▼ Car {name: 'Audi', year: 2019} ℹ
    name: "Audi"
    year: 2019
  ▼ [[Prototype]]: Object
    ▶ constructor: class Car
    ▶ [[Prototype]]: Object
```

Constructor is a special method of a class for creating and initializing an object instance of class.

5. **Class Inheritance:** To create a class inheritance, use the **extends** keyword.
A class created with a class inheritance inherits all the methods from another class.

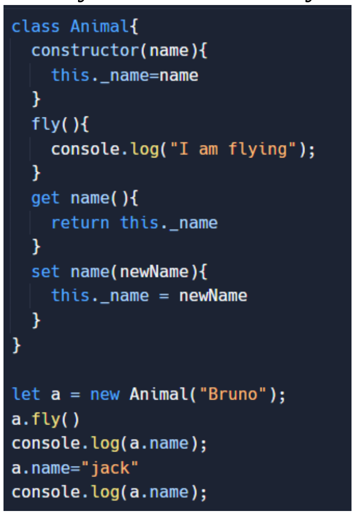


```
class Animal{
  constructor(name,color){
    this.name = name
    this.color = color
  }
  run(){
    console.log(this.name+" is Running");
  }
  shout(){
    console.log(this.name+" is Shouting");
  }
}

class Rabbit extends Animal{
  eatBanana(){
    console.log(this.name+" is eating banana");
}
}
let ani= new Animal("Elephant","black");
let rab= new Rabbit("Bunny","white");

ani.run();
ani.shout();
rab.eatBanana();
rab.run();
```

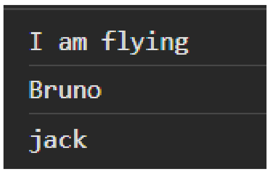


```
Elephant is Running
Elephant is Shouting
Bunny is eating banana
Bunny is Running
```

➤ **Getters and setters:**
A **getter** is a method that gets the value of a specific property. It is defined using the get keyword followed by the property name.
A **setter** is a method that sets the value of a specific property. It is defined using the set keyword followed by the property name.



```
class Animal{
  constructor(name){
    this._name=name
  }
  fly(){
    console.log("I am flying");
  }
  get name(){
    return this._name
  }
  set name(newName){
    this._name = newName
  }
}

let a = new Animal("Bruno");
a.fly()
console.log(a.name);
a.name="jack"
console.log(a.name);
```

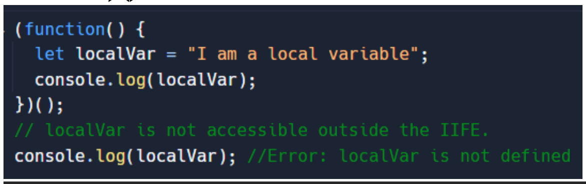


```
I am flying
Bruno
jack
```

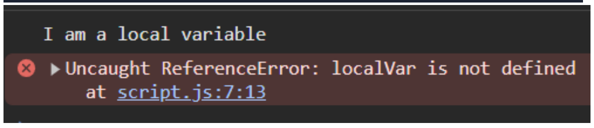
ADVANCED JAVASCRIPT

1. **IIFE**: IIFE stands for "Immediately Invoked Function Expression." It is a JavaScript function that is defined and executed immediately after its creation. The primary purpose of an IIFE is to create a new scope for variables, helping to avoid polluting the global namespace.
   Syntax:

   (function() {
   // code to be executed immediately
   })();

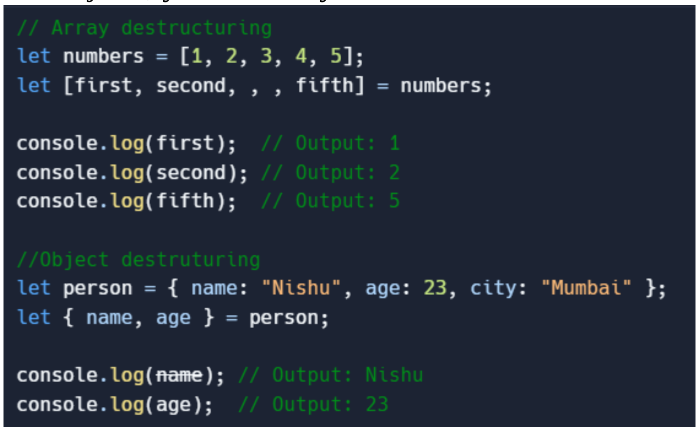


```
(function() {
  let localVar = "I am a local variable";
  console.log(localVar);
})();
// localVar is not accessible outside the IIFE.
console.log(localVar); //Error: localVar is not defined
```



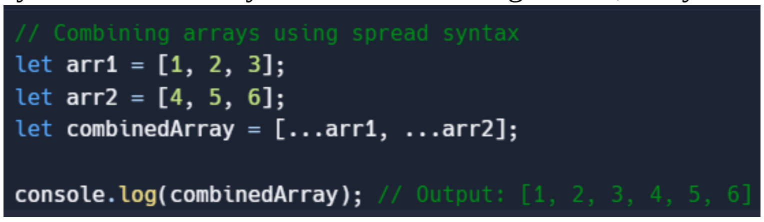
```
  I am a local variable
⊗ ▸Uncaught ReferenceError: localVar is not defined
      at script.js:7:13
```

2. **Destructuring:** It is a feature that allows you to extract values from arrays or properties from objects and assign them to variables in a more concise and expressive way. It can make your code cleaner and more readable. Destructuring is commonly used in variable assignment, function parameters, and iterating over data structures.
   You can destructure values from an array using square brackets **[]** on the left side of the assignment.
   For objects, you use curly braces **{}** on the left side of the assignment.



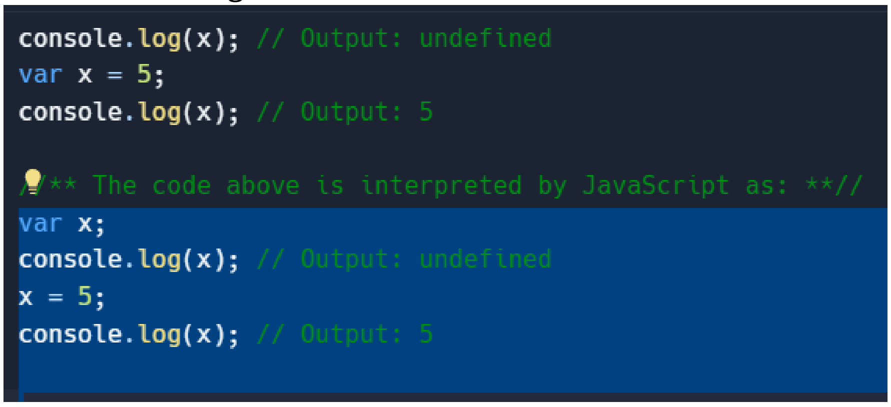
```
// Array destructuring
let numbers = [1, 2, 3, 4, 5];
let [first, second, , , fifth] = numbers;

console.log(first);  // Output: 1
console.log(second); // Output: 2
console.log(fifth);  // Output: 5

//Object destruturing
let person = { name: "Nishu", age: 23, city: "Mumbai" };
let { name, age } = person;

console.log(name); // Output: Nishu
console.log(age);  // Output: 23
```

3. **Spread Syntax:** The spread syntax (...) in JavaScript is used to expand elements of an iterable (like an array or a string) or properties of an object. It provides a concise way to copy elements or properties from one iterable or object to another. The spread syntax is commonly used in function arguments, array literals, and object literals.



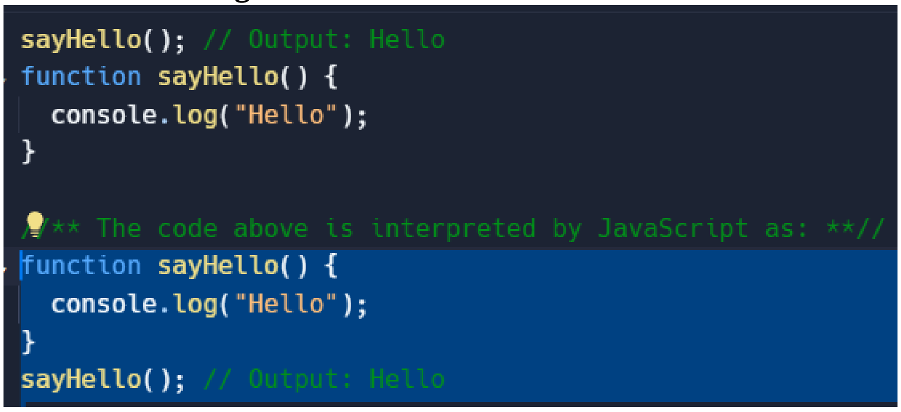
```
// Combining arrays using spread syntax
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let combinedArray = [...arr1, ...arr2];

console.log(combinedArray); // Output: [1, 2, 3, 4, 5, 6]
```

4. **Hoisting:** Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed. This means that you can use a variable or call a function before it is declared in the code.
   - **Variable Hoisting:**



```
console.log(x); // Output: undefined
var x = 5;
console.log(x); // Output: 5

/** The code above is interpreted by JavaScript as: **//
var x;
console.log(x); // Output: undefined
x = 5;
console.log(x); // Output: 5
```

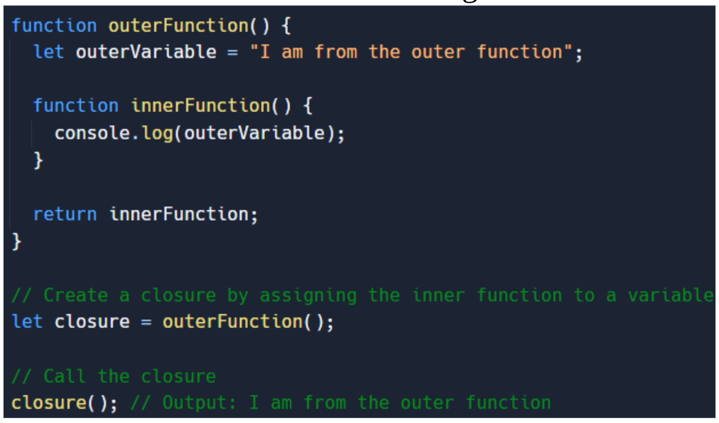
   - **Function Hoisting:**



```
sayHello(); // Output: Hello
function sayHello() {
  console.log("Hello");
}

/** The code above is interpreted by JavaScript as: **//
function sayHello() {
  console.log("Hello");
}
sayHello(); // Output: Hello
```

   Variables defined with let and const are hoisted to the top of the block, but not initialized.
   Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.
   Using a let variable before it is declared will result in a **ReferenceError.**

5. **Closure:** A closure is a fundamental concept in JavaScript that allows functions to maintain access to variables from their outer (enclosing) lexical scope, even after the outer function has finished executing.

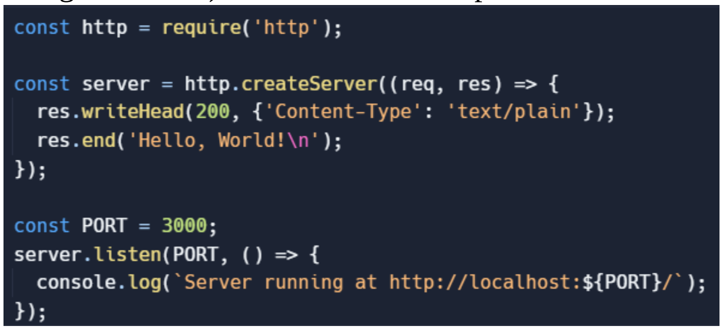


```
function outerFunction() {
  let outerVariable = "I am from the outer function";

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

  return innerFunction;
}

// Create a closure by assigning the inner function to a variable
let closure = outerFunction();

// Call the closure
closure(); // Output: I am from the outer function
```

JAVASCRIPT ON SERVER

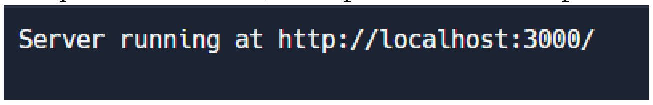
JavaScript is traditionally associated with client-side scripting, running in web browsers to enhance the interactivity of web pages. However, with the introduction of Node.js, JavaScript can also be used on the server side to build scalable and efficient web applications.

➢ **Node.js:**
Node.js is a server-side JavaScript runtime that allows developers to use JavaScript for server-side scripting. It uses the V8 JavaScript runtime engine (the same engine used by Google Chrome) to execute JavaScript code on the server.



```
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, World!\n');
});

const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}/`);
});
```

In this example, Node.js is used to create an HTTP server that listens on port 3000. When a request is received, it responds with a simple "Hello, World!" message.

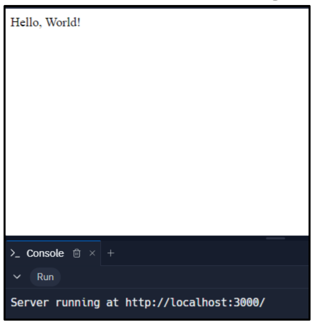


```
Server running at http://localhost:3000/
```

➢ **npm:** node package manager. Used to install packages.
Common npm Commands:

- **npm install**: Install project dependencies.
- **npm install -g:** Install a package globally.
- **npm init:** Creates “package.json” file which contains metadata about the project.
- **npm uninstall package-name**: Uninstall a package.
- **npm update:** Update packages to their latest versions.
- **npm outdated:** Check for outdated packages.

➢ **Express.js:** Express.js is a popular web application framework built on top of Node.js. It simplifies the process of building robust and scalable web applications by providing a set of features for routing, middleware, and handling HTTP requests and responses.



Hello, World!

```
Server running at http://localhost:3000/
```

Npm commands:

- **npm install express**
- **npm i -g nodemon**

CONCLUSION

JavaScript is a versatile and widely-used programming language that plays a central role in web development. Initially designed to enhance the interactivity of web pages, JavaScript has evolved into a powerful language that can be used for both client-side and server-side development. Its asynchronous and event-driven nature, along with features like closures, modules, and various frameworks, contribute to its flexibility and scalability.

On the client side, JavaScript is executed by web browsers, allowing developers to create dynamic and responsive user interfaces. With the introduction of modern frameworks and libraries such as React, Angular, and Vue.js, developers can efficiently build complex and interactive web applications.

On the server side, Node.js enables the use of JavaScript for server scripting, providing a consistent language throughout the entire development stack. This allows for code reuse, maintenance efficiency, and the creation of real-time applications.

Additionally, the JavaScript ecosystem is enriched by a vibrant package management system, npm, which facilitates the sharing and distribution of reusable code in the form of packages or modules. Developers can leverage a wide range of libraries and tools from the npm registry to streamline their development process.

While JavaScript has come a long way since its inception, it continues to evolve with the latest ECMAScript standards, introducing new language features and improvements. Its adaptability, community support, and widespread use make JavaScript a fundamental language in the world of web development. Whether building interactive user interfaces, server-side applications, or cross-platform mobile apps, JavaScript remains a cornerstone technology in the modern software development landscape.