JavaScript

Create Your Code Masterpiece for Web Development Success

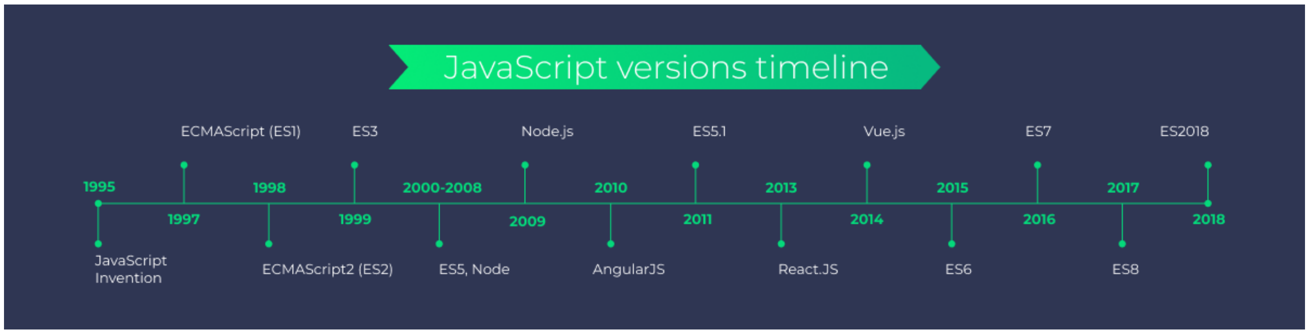
WHAT IS JAVASCRIPT ?

JavaScript is a programming language for making websites interactive. It works alongside HTML and CSS, adding dynamic features like form validation and updating content. It's versatile and widely used for web development.

JavaScript History :

JavaScript was invented by Brendan Eich in 1995. a Netscape programmer developed a new scripting language in just 10 days. It was originally named Mocha, but quickly became known as LiveScript and, later, JavaScript.

Timeline of JavaScript :



WHY WE NEED JAVASCRIPT ?

JavaScript is crucial for making websites interactive and dynamic. It adds features like form validation and content updates, enhancing the overall user experience. Without JavaScript, websites would be static and less engaging.

HOW TO USE THE JAVASCRIPT ?

We can use the JavaScript in three different ways along with our html.

1. Inline JavaScript

2. Internal JavaScript

3. External JavaScript

1.Inline JavaScript :

Inline JavaScript is used in the same line similar to inline CSS though, we have to use the functions at which point we need to activate the JavaScript. We will in detail when we are discussing about it.

Syntax :

```
< tagName  function = “ javaScript code” > </tagName>
```

Example :

```
<button onclick = “ function count() { counter++;
console.log(counter);}”>

Increase

</button>
```

2. Internal JavaScript :

Internal JavaScript is used in the same file, we separate it by using the “ script ” tag.

Syntax :

```
<script>

//javascript code

</script>
```

Example :

```
<script>
function count(){
counter++;
console.log(counter);
}
```

3.External JavaScript :

External javascript is mainly used for it's reusablility nature. In external javascript we write the javascript code in a seperate file and will include to the necessary html document

Syntax :

```
<script src = "path"> </script>
```

Example :

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

NOTE : To link the CSS file we use the link tag though, for Javascript we use the script

Basic structure for javascript:

Unlike C, java we will not have any structure to the javascript. But we have to write the code in a systematic way.

Output in javascript :

1. console

When we are using the console we will display the output inside the browser console.

There are different styles to use the console to display output :

1. log

2. info

3. error

4. warn

Syntax :

```
console.type("content");
```

2. alert :

When we are using the alert then it will show us a alert box on the browser.

Syntax :

```
alert("content"); or windows.alert("content");
```

Comments :

In javascript we are having comments. Comments are non-executable code which will help us to write some info which are not belong to the code to make others understand.

There are two types of comments in javascript:

1. single line comments

2. multi line comments

1. single line comments :

We can only write one line of comments in single line comments.

Syntax:

```
// single line comments
```

2. Multi-line comments :

We can write multiple lines of comments in multi line comments.

Syntax :

```
/*

multiple lines of comments

*/
```

Variables in Javascript:

Variables are containers which will store values inside them temporily. To name a variable we need to follow some rules.

Rules :

1. Declare the variable :

We have the declare a variable 1st inorder to access it.

We are having the 3 types of declarations.

a. let - can change based on value/ data

b. const - can't change value in the entire the program

c. Var - Var is similar to let but it is functional level variable.

NOTE : We use the let instead of var as it is more flexiable and will not allow any leakages that will happen with var.

Syntax :

declarationTyle varName;

Note : VarName should be unique, We cannot declare the same variable twice.

Datatypes in javascript :

Datatypes represents the type of data we are storing inside the variable. JavaScript has several built-in data types that are used to represent different kinds of values. Here are the main data types in JavaScript:

1. **Primitive Data Types:**

- ***Undefined:*** Represents an uninitialized or undefined value.

Syntax :

let undefinedVar;

console.log(undefinedVar); // Output: undefined

- ***Null:*** Represents the absence of any object value.

Syntax :

```
let nullVar = null;
console.log(nullVar);  // Output: null
```

- **Boolean:** Represents a logical entity and can have two values: `true` or `false`.

Syntax :

```
    let isTrue = true;
    console.log(isTrue);  // Output: true
```

- **Number:** Represents numeric values, including integers and floating-point numbers.

Syntax :

```
let integerNumber = 42;
let floatingPointNumber = 3.14;
```

- **String:** Represents sequences of characters, enclosed within single or double quotes.

Syntax :

```
let stringVar = "Hello";
console.log(stringVar);  // Output: Hello
```

- **Symbol:** Introduced in ECMAScript 6, symbols are unique and immutable primitive.

values, often used as property keys in objects.

Syntax :

```
let symbol1 = Symbol('uniqueSymbol');
        let symbol2 = Symbol('uniqueSymbol');
```

```
console.log(symbol1 === symbol2);  // Output: false (symbols are
always unique)
```

2. **Object:**

- *****Object:***** A compound data type that allows you to group related data and functions together.

Example :

```
let person = {
          firstName: 'John',
         lastName: 'Doe',
         age: 30,
        isStudent: false,
};
```

3. **Special Objects:**

- *****Function:***** A subtype of objects that is callable as a function.

Example :

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

- *****Array:***** A subtype of objects that represents an ordered collection of values.

Example :

```
let numbers = [1, 2, 3, 4, 5];
let fruits = ['apple', 'orange', 'banana'];
```

- *****Date:***** Represents a specific point in time, with associated methods for working with dates and times.

Example :

let currentDate = new Date();

- *****RegExp (Regular Expression):***** Represents a regular expression for pattern matching.

  Example :

  let pattern = /abc/;

- *****Map:***** A collection of key-value pairs where keys can be of any data type.

  Example :

  let myMap = new Map();

  myMap.set('key1', 'value1');

  myMap.set('key2', 'value2');

- *****Set:***** A collection of unique values.

  Example :

  let mySet = new Set([1, 2, 3, 4, 5]);

4. **Primitive Wrapper Objects (Non-primitive data types):**

- *****String Object:***** A wrapper object for primitive string values.

let stringObject = new String('This is a string object');

- *****Number Object:***** A wrapper object for primitive number values.

let numberObject = new Number(42);

- *****Boolean Object:***** A wrapper object for primitive boolean values.

let booleanObject = new Boolean(true);

Note : It's important to note that JavaScript is a dynamically typed language, meaning the data type of a variable is not explicitly declared but is inferred at runtime. Additionally, JavaScript is loosely typed, allowing values to be coerced from one type to another in certain situations.

Operators in javascript :

We use the operators in javascript to perform specific operations with the help of operators. In javascript we are having 7 operators.

Here are some of the key types of operators in JavaScript:

1. **Arithmetic Operators:**

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` (Division)
- `%` (Modulus - remainder of division)

Example :

```
let a = 5;
let b = 2;
console.log(a + b);  // Output: 7
```

2. **Assignment Operators:**

- `=` (Assignment)
- `+=`, `-=`, `*=`, `/=` (Compound assignment)

Example :

```
let x = 10;
x += 5; // Equivalent to x = x + 5;
```

3. **Comparison Operators:**

- `==` (Equality, loose equality)
- `===` (Strict equality)
- `!=` (Inequality, loose inequality)
- `!==` (Strict inequality)
- `>`, `<`, `>=`, `<=` (Greater than, Less than, Greater than or equal, Less than or equal)

Example :

```
let num1 = 5;
let num2 = '5';
console.log(num1 == num2);   // Output: true (loose equality)
console.log(num1 === num2);  // Output: false (strict equality)
```

4. **Logical Operators:**

- `&&` (Logical AND)
- `||` (Logical OR)
- `!` (Logical NOT)

Example :

```
let isTrue = true;
let isFalse = false;
console.log(isTrue && isFalse);  // Output: false (logical AND)
console.log(isTrue || isFalse);  // Output: true (logical OR)
console.log(!isTrue);            // Output: false (logical NOT)
```

5. **Increment/Decrement Operators:**

- `++` (Increment)
- `--` (Decrement)

Example :

```
let count = 10;
count++;
```

6. **Concatenation Operator:**

- `+` (String concatenation)

Example :

```
let str1 = 'Hello';
let str2 = 'World';
```

```
                  let greeting = str1 + ' ' + str2;  // Output: 'Hello World'
```

7. **Conditional (Ternary) Operator:**

- `condition ? expr1 : expr2`;

Example :

```
  let age = 20;
  let message = (age >= 18) ? 'Adult' : 'Minor';
```

Accept user Input :

To accept user input we follow two ways.

1. To use the prompt
2. To use the text box to submit the data.

1. Prompt :

prompt belongs to windows. hence we use the below syntax to accept the user input from the browser itself. It accepts the strings by default.

Syntax :

```
variableName = window.prompt(" message  ");
```

2. Using textbox :

For that we need to add the onclick method inorder to accept the data.

Example :

```
document.getElementById("IdName-button").onclick = function(){
variable = document.getElementById("idName-input").value;
}
```

Note : If we need to change the datatype of variable we will use the type converstion.

Type Converstion :

Type converstion is used to convert a value from one datatype to another.

- To type convert we need to use the datatype before the value inorder to change it's datatype. We use the below function inorder to change the datatype

Number( ) - to convert into number

String( ) - to convert into String

Boolean( ) - to convert into Boolean

Syntax :

varName = dataType(varName/ value);

Example :

let a = '2'; // String

a = Number(a); // converting from String to Number

NOTE : if we are trying to convert any variable which is not initialized then it will convert them using the default values.

1. NaN - not a number > for nuber

2. undefined - > for values which are not defined

3. false - > for Boolean

Math Lib in javascript :

We can use the mathematical operations using the math lib. In JavaScript, the Math object provides a set of built-in mathematical functions and constants. Here are some commonly used functions and constants available in the Math object:

### Mathematical Constants:

1. **`Math.PI`**: Represents the mathematical constant Pi (approximately 3.14159).

2. **`Math.E`**: Represents the mathematical constant Euler's number (approximately 2.71828).

### Basic Mathematical Functions:

1. **`Math.abs(x)`**: Returns the absolute value of a number.

2. **`Math.ceil(x)`**: Rounds a number up to the nearest integer.

3. **`Math.floor(x)`**: Rounds a number down to the nearest integer.

4. **`Math.round(x)`**: Rounds a number to the nearest integer.

5. **`Math.max(x, y, ...)`**: Returns the highest value among the given arguments.

6. **`Math.min(x, y, ...)`**: Returns the lowest value among the given arguments.

### Exponential and Logarithmic Functions:

1. **`Math.exp(x)`**: Returns the value of Euler's number raised to the power of x.

2. **`Math.log(x)`**: Returns the natural logarithm (base e) of a number.

3. **`Math.log10(x)`**: Returns the base 10 logarithm of a number.

### Trigonometric Functions:

1. **`Math.sin(x)`**: Returns the sine of an angle (in radians).

2. **`Math.cos(x)`**: Returns the cosine of an angle (in radians).

3. **`Math.tan(x)`**: Returns the tangent of an angle (in radians).

4. **`Math.asin(x)`**: Returns the arcsine of a number, returning values in the range of -π/2 to π/2 radians.

5. **`Math.acos(x)`**: Returns the arccosine of a number, returning values in the range of 0 to π radians.

6. **`Math.atan(x)`**: Returns the arctangent of a number, returning values in the range of -π/2 to π/2 radians.

### Random Number Functions:

1. **`Math.random()`**: Returns a pseudo-random number between 0 (inclusive) and 1 (exclusive).

2. **`Math.floor(Math.random() * (max - min + 1)) + min`**: Generates a random integer between min (inclusive) and max (inclusive).

Conditional Statments :

Conditional statements in JavaScript are used to make decisions in your code based on certain conditions. The most common conditional statements in JavaScript are:

1. **if statement:**

The `if` statement is used to execute a block of code if a specified condition evaluates to true.

Syntax :

```
if (condition) {
   // code to be executed if the condition is true
}
```

2. **if-else statement:**

The `if-else` statement allows you to execute one block of code if the condition is true and another block if the condition is false.

Syntax :

```
if (condition) {
   // code to be executed if the condition is true
} else {
   // code to be executed if the condition is false
}
```

3. **if-else if-else statement:**

You can use multiple conditions with the `if-else if-else` statement to check for different cases.

Syntax :

```
if (condition1) {
   // code to be executed if condition1 is true
} else if (condition2) {
   // code to be executed if condition2 is true
} else {
        // code to be executed if none of the conditions are true
    }
```

4. **switch statement:**

The `switch` statement is useful when you have multiple possible conditions to check. It's an alternative to using multiple `if-else if` statements.

Syntax :

```
switch (expression) {
    case value1:
        // code to be executed if expression matches value1
        break;
    case value2:
        // code to be executed if expression matches value2
        break;
    // ... more cases
    default:
        // code to be executed if expression doesn't match any case
}
```

Strings in javaScript :

In JavaScript, strings are sequences of characters, such as text, and they are used to represent and manipulate textual data. Strings can be created using single or double quotes, and there are various methods available to perform operations on strings. Here are some basics about strings in JavaScript:

### Creating Strings:

You can create strings using single or double quotes:

Example :

```
let singleQuotedString = 'This is a single-quoted string.';
let doubleQuotedString = "This is a double-quoted string.";
```

Both single and double quotes are acceptable, and you can use them interchangeably. This flexibility can be useful in certain situations.

### String Concatenation:

You can concatenate (combine) strings using the `+` operator:

Example :

```
let firstName = "John";
let lastName = "Doe";
let fullName = firstName + " " + lastName;
console.log(fullName);  // Output: John Doe
```

### String Length:

You can find the length of a string using the `length` property:

Example :

```
let message = "Hello, World!";
let messageLength = message.length;
console.log(messageLength);  // Output: 13
```

### Accessing Characters:

Individual characters in a string can be accessed using square brackets and the character's index (zero-based):

Example :

```
let str = "JavaScript";
let firstChar = str[0];  // J
let thirdChar = str[2];  // v
```

### String Methods:

JavaScript provides various methods for working with strings. Some common ones include:

- `toUpperCase()` and `toLowerCase()`:

Convert a string to uppercase or lowercase.

Example :

```
        let text = "Hello, World!";
        let uppercased = text.toUpperCase();   // HELLO,  WORLD!
    let lowercased = text.toLowerCase();   // hello, world!
```

- `indexOf()`:

Find the index of a substring within a string.

-lastIndexof() - used to return the last index o f the value/ character.

Example :

```
         let sentence = "JavaScript is awesome!";
         let index = sentence.indexOf("awesome");   // 15
```

- `slice(start, end)`:

Extract a portion of a string.

Example :

```
         let phrase = "To be or not to be";
         let sliced = phrase.slice(6, 13);  // "or not"
```

- `replace(oldString, newString)`:

Replace occurrences of a substring with another.

Example :

```
         let original = "I like cats.";
         let modified = original.replace("cats",  "dogs");  // "I like dogs."
```

-trim() :

trim() method is used to remove whitespace (spaces, tabs, and newlines) from both ends of a string.

Example :

```
        let stringWithWhitespace = "   Hello, World!   ";
        // Using trim() to remove leading and trailing whitespace
        let trimmedString = stringWithWhitespace.trim();
        console.log(trimmedString);  // Output: "Hello, World!"
```

-repeat() :

Used to repeat the String n- number of times.

Example :

```
let a = "Hello";
console.log(a.repeat(5)); // "HelloHelloHelloHelloHello"
```

- startsWith() and endsWith() :

Used to check where the string starts/end with the give info or not.

Example :

```
let varname = "String";
varname.startsWith('D'); // false
varname.endsWith('h'); //true
```

-includes():

The includes() method is a built-in method of the JavaScript String and Array objects. It is used to check whether a particular value (substring for strings or element for arrays) is present in the given string or array.

Example :

```
let text = "Hello, World!";
        // Check if the string includes a specific substring
        let containsHello = text.includes("Hello");
        console.log(containsHello);  // Output: true
        let containsFoo = text.includes("Foo");
        console.log(containsFoo);  // Output: false
```

-replaceAll() :

It is used to replace all occurrences of a specified substring or regular expression with another string. This method is an improvement over the older replace() method, which only replaced the first occurrence by default.

Example :

```
let originalString = "Hello, World! Hello, Universe!";
    let newString = originalString.replaceAll("Hello", "Hola");
    console.log(newString);// Output: "Hola, World! Hola, Universe!"
```

-replace() :

It is used to replace the first occurrence only.

Example :

```
// Using replace() to replace only the first occurrence
    let firstReplacement = originalString.replace("Hello", "Hola");
    console.log(firstReplacement); / Output: "Hola, World! Hello,
    Universe!"
```

-padStart() :

The padStart() method is a string method in JavaScript that allows you to pad the beginning of a string with a specified number of characters until the desired length is reached. This method is useful for aligning strings, especially in cases where you want to ensure a minimum length for a string.

Example :

```
let originalString = "42";
let paddedString = originalString.padStart(5, "0");
console.log(paddedString);// Output: "00042"
```

-padEnd() :

The padEnd() method is a string method in JavaScript that allows you to pad the ending of a string with a specified number of characters until the desired length is reached. This method is useful for aligning strings, especially in cases where you want to ensure a minimum length for a string.

Example :

```
let originalString = "42";
let paddedString = originalString.padEnd(5, "0");
console.log(paddedString);// Output: "42000"
```

-slice() :

Slicing method is used to create a sub string from the original string. String slicing follows the below syntax :

Syntax :

```
stringVar.slice(IndexStart, IndexEnd);
```

Example :

```
let originalString = "I like JavaScript";
console.log(originalString.slice(2, 8); // like J
```

Loops in javaScript :

In JavaScript, loops are used to execute a block of code repeatedly until a specified condition is met. There are several types of loops in JavaScript, each serving different purposes. Here are the main types of loops:

### 1. `for` Loop:

The `for` loop is commonly used when the number of iterations is known in advance.

Syntax :

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

Example:

```
for (let i = 0; i < 5; i++) {
   console.log(i); // Output: 0, 1, 2, 3, 4
}
```

### 2. `while` Loop:

The `while` loop is used when you want to execute a block of code as long as a specified condition is true.

Syntax :

```
while (condition) {
   // code to be executed as long as the condition is true
}
```

Example:

```
let i = 0;
while (i < 5) {
   console.log(i); // Output: 0, 1, 2, 3, 4
   i++;
}
```

### 3. `do-while` Loop:

The `do-while` loop is similar to the `while` loop, but it ensures that the block of code is executed at least once before checking the condition.

Syntax :

```
do {
   // code to be executed
} while (condition);
```

Example:

```
let i = 0;
do {
   console.log(i); // Output: 0 (even though the condition is false)
   i++;
} while (i < 0);
```

### Loop Control Statements:

JavaScript also provides loop control statements such as `break` and `continue`:

- *The `break` statement is used to exit a loop prematurely.*

- *The `continue` statement is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration.*

Example using `break`:

```
for (let i = 0; i < 5; i++) {
   if (i === 3) {
      break; // exit the loop when i is 3
   }
   console.log(i); // Output: 0, 1, 2
}
```

Example using `continue`:

```
for (let i = 0; i < 5; i++) {
   if (i === 2) {
      continue; // skip the rest of the code for i = 2
   }
   console.log(i); // Output: 0, 1, 3, 4
}
```

Function :

Functions in JavaScript are blocks of reusable code that can be defined and called to perform a specific task. Functions help organize code, promote reusability, and make it easier to maintain and debug programs. Here's an overview of how functions work in JavaScript:

### 1. Function Declaration:

You can declare a function using the `function` keyword, followed by the function name, a list of parameters (if any), and the code block.

Example :

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

### 2. Function Expression:

You can also define a function using a function expression, where the function is assigned to a variable.

Example :

```
let greet = function(name) {
    console.log("Hello, " + name + "!");
};
// Call the function
greet("Jane"); // Output: Hello, Jane!
```

### 3. Arrow Functions (ES6+):

Arrow functions provide a concise syntax for defining functions, especially for short functions with a single statement.

Example :

```
let greet = (name) => {
    console.log("Hello, " + name + "!");
};
// Call the function
```

```
greet("Alice"); // Output: Hello, Alice!
```

### 4. Function Parameters and Return Values:

Functions can take parameters, which are values passed to the function, and they can return a value using the `return` keyword.

Example :

```
function add(a, b) {
    return a + b;
}
let result = add(3, 4);
console.log(result); // Output: 7
```

### 5. Default Parameters (ES6+):

ES6 introduced default parameter values, allowing you to provide default values for function parameters.

Example :

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

### 6. Function Scope:

Variables declared inside a function are local to that function, creating a function scope. They are not accessible outside the function.

Example :

```
function example() {
    let localVar = "I am a local variable";
    console.log(localVar);
}
```

```
example(); // Output: I am a local variable
// console.log(localVar); // This would result in an error
```

### 7. Function Invocation:

Functions can be invoked (called) in various ways, including as standalone functions, as methods of objects, or using the `call()` and `apply()` methods.

Example :

```
// Standalone function
function sayHello() {
    console.log("Hello!");
}
sayHello(); // Output: Hello!
// Function as a method
let obj = {
    greet: function() {
        console.log("Greetings!");
    }
};
obj.greet(); // Output: Greetings!
```

Arrays :

In JavaScript, an array is a data structure that allows you to store and organize multiple values in a single variable. Arrays can hold elements of any data type, including numbers, strings, objects, and other arrays. The elements in an array are indexed starting from 0, and you can access them using square brackets.

### Creating Arrays:

Arrays can be created using the `Array` constructor or using array literals.

#### Using Array Literals:

Example :

```
// Array of numbers
let numbers = [1, 2, 3, 4, 5];
// Array of strings
let fruits = ["Apple", "Banana", "Orange"];
// Array of mixed data types
let mixedArray = [1, "two", true, { key: "value" }];
```

### Accessing Elements:

You can access elements in an array using their index. Remember that array indices start from 0.

Example :

```
let fruits = ["Apple", "Banana", "Orange"];
console.log(fruits[0]); // Output: Apple
console.log(fruits[1]); // Output: Banana
```

### Modifying Elements:

You can modify elements in an array by assigning new values to specific indices.

Example :

```
let numbers = [1, 2, 3, 4, 5];
numbers[2] = 10;
console.log(numbers); // Output: [1, 2, 10, 4, 5]
```

### Array Methods:

JavaScript provides a variety of built-in methods for working with arrays. Here are a few examples:

#### a. `push()` and `pop()`:

Example :

```
let numbers = [1, 2, 3];
```

```
numbers.push(4); // Add an element to the end
console.log(numbers); // Output: [1, 2, 3, 4]
let poppedElement = numbers.pop(); // Remove and return the last element
console.log(numbers); // Output: [1, 2, 3]
console.log(poppedElement); // Output: 4
```

#### b. `shift()` and `unshift()`:

Example :

```
let fruits = ["Banana", "Orange", "Apple"];
fruits.shift(); // Remove the first element
console.log(fruits); // Output: ["Orange", "Apple"]
fruits.unshift("Mango"); // Add an element to the beginning
console.log(fruits); // Output: ["Mango", "Orange", "Apple"]
```

#### c. `slice()`:

Example :

```
let numbers = [1, 2, 3, 4, 5];
let slicedArray = numbers.slice(1, 4); // Extract elements from index 1 to 3 (not including 4)
console.log(slicedArray); // Output: [2, 3, 4]
```

#### d. `splice()`:

Example :

```
let numbers = [1, 2, 3, 4, 5];
numbers.splice(2, 2, 10, 11); // Remove 2 elements starting from index 2 and insert 10 and 11
console.log(numbers); // Output: [1, 2, 10, 11, 5]
```

For each method :

If you are looking to use a callback function with an array iteration, the `forEach` method in JavaScript is a commonly used approach.

For each method will uses the following arguments :

1. ***currentValue:*** The current element being processed in the array.
2. ***index (optional):*** The index of the current element being processed.
3. ***array (optional):*** The array that forEach is being applied to.

Syntax :

```
array.forEach(function(currentValue, index, array) {
      // Code to be executed for each element
     });
```

Example :

```
let fruits = ['apple', 'banana', 'orange'];
     fruits.forEach(function(fruit, index, array) {
      console.log(`Index ${index}: ${fruit}`);
     });
```

Map() :

The `map()` method in JavaScript is used to create a new array by applying a provided function to each element of the original array.

Syntax :

```
const newArray = array.map(function(currentValue, index, array) {
 // Code to be executed for each element
 // Return the new element to be included in the new array
 return newElement;
});
```

- *`currentValue`: The current element being processed in the array.*

- *`index` (optional): The index of the current element being processed.*
- *`array` (optional): The array that `map` is being applied to.*

Example :

```
let numbers = [1, 2, 3, 4, 5];
let squaredNumbers = numbers.map(function(number) {
  return number * number;
});
console.log(squaredNumbers);  // Output: [1, 4, 9, 16, 25]
```

In this example, the `map` method creates a new array `squaredNumbers` by squaring each element of the original `numbers` array.

You can also use arrow functions for a more concise syntax:

Example :

```
let numbers = [1, 2, 3, 4, 5];
let squaredNumbers = numbers.map(number => number * number);
console.log(squaredNumbers);  // Output: [1, 4, 9, 16, 25]
```

NOTE : The `map()` method is useful when you want to transform each element of an array and create a new array based on those transformations.

.filter() :

In JavaScript, you can use the `filter` method to create a new array with elements that pass a certain condition. The `filter` method is available for arrays and takes a callback function as its argument. This callback function is applied to each element of the array, and if the function returns `true`, the element is included in the new array; otherwise, it is excluded.

Syntax :

```
const newArray = originalArray.filter(function(element, index, array) {
    // Your filtering condition goes here
    // Return true if the element should be included in the new array, false
    otherwise
});
// or using arrow function for a more concise syntax:
const newArray = originalArray.filter((element, index, array) => {
    // Your filtering condition goes here
    // Return true if the element should be included in the new array, false
    otherwise
});
```

Explanation:

- `originalArray`: The array you want to filter.
- `element`: The current element being processed in the array.
- `index`: The index of the current element in the array.
- `array`: The array being processed.

Inside the callback function, you provide the condition that determines whether an element should be included in the new array. If the condition is met (returns `true`), the element is included; otherwise, it is excluded. The resulting filtered array is stored in `newArray`.

Example :

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
// Filtering even numbers
const evenNumbers = numbers.filter(function(number) {
  return number % 2 === 0;
});
console.log(evenNumbers); // Output: [2, 4, 6, 8, 10]
```

You can also use arrow functions for a more concise syntax:

Example :

```
// Using arrow function for filtering odd numbers

const oddNumbers = numbers.filter(number => number % 2 !== 0);

console.log(oddNumbers); // Output: [1, 3, 5, 7, 9]
```

In these examples, the `filter` method is applied to an array of numbers, and it creates a new array containing only even numbers or odd numbers, depending on the condition specified in the callback function.

Reduce() :

In JavaScript, the `reduce` method is used to reduce an array to a single value. It takes a callback function as its argument, which is applied to each element of the array in sequence to accumulate a result.

Syntax :

```
const result = array.reduce(function(accumulator, currentValue, currentIndex, array) {

  // Your reduction logic goes here

  return updatedAccumulator; // This value will be used as the accumulator in the next iteration

}, initialValue);
```

- *`accumulator`: The accumulated result of the reduction.*

- *`currentValue`: The current element being processed in the array.*

- *`currentIndex`: The index of the current element in the array.*

- *`array`: The array being processed.*

- *`initialValue` (optional): An initial value for the accumulator. If not provided, the first element of the array is used as the initial accumulator value.*

Example :

```
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce(function(accumulator, currentValue) {

  return accumulator + currentValue;
```

```
    }, 0);

    console.log(sum); // Output: 15
```

In this example, the `reduce` method is used to add up all the numbers in the array. The initial value of the accumulator is set to `0`, and the callback function simply adds the current element to the accumulator in each iteration.

Spread Operator :

It is used to unpack the arrays. The spread operator is represented by '...' infront of variable, the spread operator (...) is a versatile syntax that allows you to expand elements of an array or properties of an object. It is commonly used for creating shallow copies of arrays and objects, merging arrays, and passing function arguments.

Syntax :

```
...arrayName;
```

Example :

```
let originalArray = [1, 2, 3];

let copyArray = [...originalArray];

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

Rest Parameters :

In JavaScript, the rest parameter, denoted by the ellipsis (`...`) followed by a parameter name, allows a function to accept any number of arguments as an array. The rest parameter collects the remaining arguments into a single array, making it useful when you want to work with a variable number of parameters.

Here's an example of using the rest parameter in a function:

```
function sum(...numbers) {

   return numbers.reduce((acc, num) => acc + num, 0);

}
```

```
console.log(sum(1, 2, 3, 4, 5)); // Output: 15

console.log(sum(10, 20, 30));    // Output: 60
```

In this example, the `...numbers` rest parameter collects all the arguments passed to the `sum` function into an array called `numbers`. The `reduce` method is then used to calculate the sum of the numbers in the array.

### Destructuring with Rest:

You can also use the rest parameter in destructuring, which allows you to extract some elements from an array and collect the remaining elements in a separate array.

Example :

```
const [first, second, ...rest] = [1, 2, 3, 4, 5];

console.log(first);  // Output: 1

console.log(second); // Output: 2

console.log(rest);   // Output: [3, 4, 5]
```

In this example, the `...rest` collects the remaining elements (3, 4, 5) into an array named `rest`.

### Rest Parameter in Function Definition:

The rest parameter can also be used in the function definition to collect a variable number of arguments into an array.

Example :

```
function example(firstArg, ...restArgs) {

   console.log(firstArg);   // Output: value1

   console.log(restArgs);   // Output: [value2, value3, value4]

}

example('value1', 'value2', 'value3', 'value4');
```

In this example, the `firstArg` parameter captures the first argument, and the `...restArgs` rest parameter captures the remaining arguments into an array.

Callback :

callback function is a function that is passed as an argument to another function and is executed after some operation has been completed.

Syntax :

```
function functionName( callback ){
// function body
callback(); // calls the next method
}
```

Example :

```
function greeting(callback){
console.log("Hola ! Good Morning "); // will display message in
                                                console
callback(); // will call the next method
}
function closing(){
console.log("Thanks for visiting. Bye "); // will display the following
                                                  message
}
greeting(closing); // calling two methods at a time.
```

Function Expression :

In JavaScript, a function expression is a way to define a function using an expression. Unlike function declarations, which are hoisted to the top of their containing scope, function expressions are not hoisted. This means that you need to define a function expression before you can use it in your code.

Syntax :

```
const myFunction = function(parameters) {
  // Function body
  // Code goes here
  return result; // Optional
};
```

- `const myFunction`: Defines a constant variable named `myFunction` that holds the function.

- `function(parameters) { /* ... */ }`: This is the function expression itself. It can take parameters, and the function body contains the code to be executed when the function is called.

- `return result;`: The `return` statement is optional and is used to specify the value that the function will return. If omitted, the function returns `undefined`.

Example :

```
const greet = function(name) {
  return `Hello, ${name}!`;
};
console.log(greet("John")); // Output: Hello, John!
```

Function expressions are often used in scenarios where you want to assign a function to a variable, pass a function as an argument to another function, or use them in immediately-invoked function expressions (IIFE).

Example :

```
const multiply = (a, b) => a * b;
console.log(multiply(2, 3)); // Output: 6
```

Arrow Functions :

Arrow functions are a concise way to write function expressions in JavaScript. They were introduced with ES6 (ECMAScript 2015) and provide a more compact syntax compared to traditional function expressions. Arrow functions are especially useful for short, one-line functions.

Syntax :

```
const myFunction = (parameter1, parameter2, ...) => {
  // Function body
  // Code goes here
  return result; // Optional
};
```

- ***`const myFunction`:*** Defines a constant variable named `myFunction` that holds the arrow function.

- ***`(parameter1, parameter2, ...) => { /* ... */ }`:*** This is the arrow function itself. It can take parameters, and the function body contains the code to be executed when the function is called.

- ***`return result;`:*** The `return` statement is optional and is used to specify the value that the function will return. If omitted, the function returns `undefined`.

Example :

```
const square = (x) => x * x;
console.log(square(5)); // Output: 25
```

If the arrow function has only one parameter, you can omit the parentheses around the parameter:

Example :

```
const greet = name => `Hello, ${name}!`;
console.log(greet("Alice")); // Output: Hello, Alice!
```

NOTE : If the function body consists of a single expression, you can omit the curly braces `{}`:

SetTimeOut :

In JavaScript, the `setTimeout()` function is used to schedule the execution of a function or the evaluation of an expression after a specified amount of time has passed. It allows you to introduce a delay in the execution of code, making it useful for scenarios like animations, asynchronous operations, or creating timed events.

Syntax :

```
setTimeout(callback, delay, param1, param2, ...);
```

- ***`callback`:*** A function or an expression to be executed after the specified delay.

- ***`delay`:*** The time (in milliseconds) to wait before executing the function or expression.

- ***`param1`, `param2`, ... (optional):*** Additional parameters to pass to the callback function.

Example :

```
function greet() {
   console.log("Hello, world!");
}
setTimeout(greet, 2000); // Execute greet() after 2000 milliseconds (2 seconds)
```

In this example, the `greet` function will be called after a delay of 2000 milliseconds (2 seconds).

### Using Anonymous Functions:

You can also use anonymous functions or arrow functions as the callback:

Example :

```
setTimeout(function() {
   console.log("Delayed message");
}, 3000);
or
setTimeout(() => {
    console.log("Delayed message");
}, 3000);
```

### **Passing Parameters:**

If you need to pass parameters to the callback function, you can include them after the delay parameter:

Example :

```
function greet(name) {
    console.log("Hello, " + name + "!");
}
setTimeout(greet, 1000, "John"); // Execute greet("John") after 1000
                                    milliseconds (1 second)
```

### **Clearing Timeout:**

The `setTimeout()` function returns a timer ID that can be used to cancel the execution of the scheduled function using the `clearTimeout()` function:

Example :

```
let timerId = setTimeout(function() {
    console.log("This won't be executed");
}, 5000);
// Cancel the scheduled execution
clearTimeout(timerId);
```

Objects :

In JavaScript, objects are a fundamental data type that allows you to group related data and functions together. Objects are instances of classes, but JavaScript uses a prototype-based inheritance model, which means objects can be created directly without the need for class definitions.

Syntax :

```
// Object literal syntax
const myObject = {
  key1: value1,
  key2: value2,
  // More key-value pairs
};
```

Each key-value pair in an object is called a property, and properties can hold various types of values, including numbers, strings, arrays, other objects, and functions.

Example :

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  hobbies: ["reading", "coding", "traveling"],
  address: {
    street: "123 Main St",
    city: "Anytown",
    country: "USA"
  },
  sayHello: function() {
    console.log(`Hello, my name is ${this.firstName}
${this.lastName}.`);
  }
};
console.log(person.firstName);  // Output: John
console.log(person.hobbies[1]); // Output: coding
```

```
person.sayHello(); // Output: Hello, my name is John Doe.
```

In this example:

- `firstName`, `lastName`, `age`, etc., are properties of the `person` object.

- The `hobbies` property is an array, and the `address` property is another object nested within the `person` object.

- The `sayHello` property is a method (function) attached to the `person` object.

You can access object properties using dot notation (`objectName.propertyName`) or bracket notation (`objectName["propertyName"]`). The latter is particularly useful when the property name contains special characters or spaces.

Example :

```
console.log(person["age"]); // Output: 30
```

You can also add, modify, or delete properties of an object dynamically:

Example :

```
person.gender = "Male"; // Adding a new property

person.age = 31; // Modifying an existing property

delete person["address"]; // Deleting a property
```

Note : The function inside the objects are called as methods.

NOTE : We can use the nested objects and also arrays in object

Array of objects :

An array of objects is a common data structure in JavaScript where each element of the array is an object. Each object in the array can have its own set of properties and values.

Example :

```
// Array of objects representing people

const people = [

  { firstName: 'John', lastName: 'Doe', age: 30 },
  { firstName: 'Alice', lastName: 'Smith', age: 25 },
  { firstName: 'Bob', lastName: 'Johnson', age: 35 }
];
// Accessing and modifying values
console.log(people[0].firstName);  // Outputs: John
console.log(people[1].age);        // Outputs: 25
// Adding a new person
const newPerson = { firstName: 'Eve', lastName: 'Williams', age: 28 };
people.push(newPerson);
// Iterating through the array
for (const person of people) {
  console.log(`${person.firstName} ${person.lastName} - Age:
${person.age}`);
}
```

In this example, the `people` array contains three objects, each representing a person with properties such as `firstName`, `lastName`, and `age`. You can perform various operations on the array, including accessing values, modifying existing objects, adding new objects, and iterating through the array.

Constructors :

In JavaScript, a constructor is a special type of function used to create and initialize objects. Constructors are typically used in conjunction with the `new` keyword to create instances of a particular type or class of objects. When a function is used as a constructor, it is intended to be called with the `new` keyword, and it is responsible for initializing the properties and methods of the new object.

Syntax :

```
function fName(parameters...);{
```

```
this.value = value;

//n -values

}
```

Example :

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
// Creating an instance of Person
const john = new Person("John", 30);
console.log(john.name); // Output: John
console.log(john.age);  // Output: 30
```

In this example:

- `Person` is a constructor function.
- The `new` keyword is used to create a new instance of `Person`.
- Inside the `Person` constructor, `this` refers to the newly created object, and properties (`name` and `age`) are assigned to it.

Constructors are often used to define classes in JavaScript, even though JavaScript itself doesn't have a native class system like some other programming languages. You can add methods to the constructor's prototype to share them among all instances created with the constructor.

Example :

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
// Adding a method to the prototype
```

```
Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.name} and I'm ${this.age} years
    old.`);
};
const john = new Person("John", 30);
john.sayHello(); // Output: Hello, my name is John and I'm 30 years old.
```

Classes :

Classes provides a more convenient way to define constructor functions and work with prototypes. Although JavaScript's class syntax resembles that of other programming languages, it's important to understand that JavaScript's classes are essentially syntactic sugar over the existing prototype-based inheritance model.

Syntax :

```
class className{
constructor(){
//constructor Code
}
methodName(){
//Method Code
}
}
```

Example :

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age}
years         old.`);
  }
}


// Creating an instance of the Person class
const john = new Person("John", 30);
john.sayHello(); // Output: Hello, my name is John and I'm 30 years old.
```

In this example:

- `class Person` declares a class named `Person`.

- `constructor(name, age)` is a special method used for initializing instances of the class. It's automatically called when a new instance is created with the `new` keyword.

- `sayHello()` is a method defined within the class.

Static :

In JavaScript, the `static` keyword is used in the context of classes to define static methods or static properties. Static members are associated with the class itself, rather than with instances of the class. They are called on the class rather than on instances of the class.

Here's how you can use the `static` keyword with classes:

### Static Methods:

Example :

```
class MathOperations {
  static add(x, y) {
    return x + y;
  }
  static subtract(x, y) {
          return x - y;
  }
}
// Calling static methods without creating an instance
console.log(MathOperations.add(5, 3));      // Output: 8
console.log(MathOperations.subtract(10, 4)); // Output: 6
```

In this example, `add` and `subtract` are static methods of the `MathOperations` class. You can call these methods directly on the class itself without creating an instance of the class.

Inheritance ;

In JavaScript, inheritance is achieved through prototype-based inheritance. While the introduction of the `class` syntax in ECMAScript 2015 (ES6) made working with inheritance more familiar to developers coming from class-based languages, it's essential to understand that JavaScript's inheritance is based on prototypes.

Example :

```
//Animal Constructor
function Animal(){
   console.log("animal");
}
// adding animalName method to Animal constructor
Animal.prototype.animalName = (name) => {
   console.log(name);
}
// child constructor
```

```
function Dog(){
   console.log("Dog");
}
//creating inheritance from Animal to Dog
Dog.prototype = Object.create(Animal.prototype);
// creating instance of Dog Constructor
const dog = new Dog();
//accessing the parent function
dog.animalName("max");
```

In this example:

- `Animal` is the parent constructor function, and we add a method `makeSound` to its prototype.
- `Dog` is the child constructor function, and it calls the `Animal` constructor using `Animal.call(this, name)` to initialize the common properties.
- `Dog.prototype` is set to an instance of `Animal.prototype` using `Object.create(Animal.prototype)`, establishing the prototype chain.

Instances of `Dog` inherit the properties and methods of both `Dog` and `Animal`.

With the introduction of the `class` syntax in ES6, achieving inheritance is more concise:

Example :

```
//creating a parent class
class Animal{
   name;
   eat(){
      console.log(this.name + " is eating");
   }
}
```

```
    // creating child class and forming an inheritance
    class Dog extends Animal{
        name = "Jack";
    }
    //creating an instance for Dog
    const dog = new Dog();
    //calling the parent constructor
    dog.eat();
```

Here, `extends` is used to declare the inheritance relationship between `Dog` and `Animal`. The `super` keyword is used to call the constructor of the parent class. This syntax is more readable and aligns with the class-based inheritance model found in other languages.

Super :

In JavaScript, the `super` keyword is used in the context of classes, and it serves two main purposes:

1. **Accessing the Parent Class:**

- In a subclass constructor, `super` is used to call the constructor of the parent class. This is necessary if the subclass has its own constructor, and it ensures that the initialization logic of the parent class is executed.

Example :

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  makeSound() {
    console.log("Some generic sound");
  }
}
class Dog extends Animal {
  constructor(name, breed) {
    // Call the constructor of the parent class
    super(name);
    this.breed = breed;
  }
  bark() {
    console.log("Woof!");
  }
}
const myDog = new Dog("Buddy", "Golden Retriever");
console.log(myDog.name);  // Output: Buddy
```

2. **Accessing the Parent Class Prototype Methods:**

- `super` is also used to call methods from the prototype of the parent class. This is particularly useful when a method with the same name exists in both the parent and subclass, and you want to specifically call the method from the parent class.

Example :

```
class Animal {
  makeSound() {
    console.log("Some generic sound");
  }
}
class Dog extends Animal {
  makeSound() {
    // Call the makeSound method from the parent class
    super.makeSound();
    console.log("Woof!");
  }
}
const myDog = new Dog();
myDog.makeSound();
// Output:
// Some generic sound
// Woof!
```

NOTE : If you want to use the parent properties or values 1st you need to call the parent constructor using the super();

TIP : We can send the parameters to the super() if you want to.

Getters and Setters :

In JavaScript, getters and setters are special methods that allow you to define how properties of an object are accessed and modified. They provide a way to control the behavior of reading and writing values to object properties. Here's a basic explanation and example:

### Getters:

A getter is a method that gets the value of a specific property.

Example :

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
};
console.log(person.fullName); // Outputs: John Doe
```

In this example, `fullName` is a getter that concatenates the `firstName` and `lastName` properties.

### Setters:

A setter is a method that sets the value of a specific property.

Example :

```
const person = {
  _age: 25, // convention: prefixing with an underscore for the private
variable
  set age(newAge) {
    if (newAge > 0 && newAge < 150) {
      this._age = newAge;
    } else {
      console.error('Invalid age value');
    }
  }
};
person.age = 30;
console.log(person._age); // Outputs: 30
person.age = 200; // Outputs: Invalid age value
```

In this example, the `age` setter checks if the new value is within a valid range before updating the `_age` property.

These getter and setter methods allow you to encapsulate the logic associated with getting and setting values, providing a level of control and validation.