JavaScript

Create Your Code Masterpiece for Web Development Success

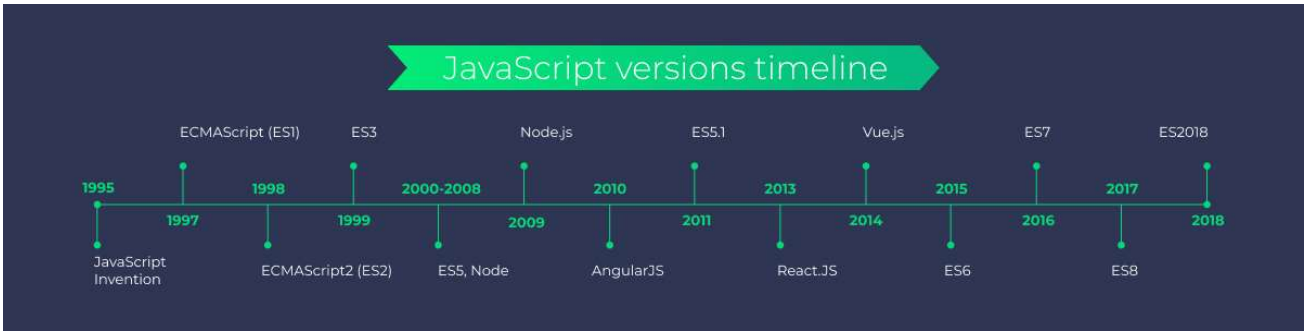
WHAT IS JAVASCRIPT ?

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.

Destructuring :

Is used to extract the values from objects or arrays to assign them to variable individually.

1. [] - used to destructure array

2.{} - used to destructure object

Array Destructuring:

Example :

```
const numbers = [1, 2, 3, 4, 5];
// Destructuring assignment
const [first, second, , fourth] = numbers;
console.log(first);  // Outputs: 1
```

```
console.log(second); // Outputs: 2
console.log(fourth); // Outputs: 4
```

Object Destructuring:

Example :

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};
// Destructuring assignment
const { firstName, lastName, age } = person;
console.log(firstName); // Outputs: John
console.log(lastName);  // Outputs: Doe
console.log(age);       // Outputs: 30
```

Renaming Variables during Destructuring:

Example :

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};
// Destructuring assignment with variable renaming
const { firstName: fName, lastName: lName, age: personAge } =
person;
console.log(fName);      // Outputs: John
```

```
console.log(lName);      // Outputs: Doe
console.log(personAge);  // Outputs: 30
```

Note : we can destructure the objects and array while passing as parameters to the functions

Closures :

A closure in JavaScript is a mechanism where a function has access to the variables from its outer (enclosing) scope, even after that outer scope has finished execution. In simpler terms, a closure allows a function to "remember" and access the variables in the scope where it was created, even if the function is executed in a different scope.

Example :

```
function outerFunction() {
  const outerVariable = 'I am from the outer function';
        function innerFunction() {
              console.log(outerVariable); // innerFunction has access to
            outerVariable
        }
  return innerFunction; // Return the inner function, creating a closure
}
const closureFunction = outerFunction();
closureFunction(); // Outputs: I am from the outer function
```

In this example:

- `outerFunction` declares a variable `outerVariable` and defines an inner function `innerFunction`.

- `outerFunction` returns `innerFunction`, creating a closure because `innerFunction` still has access to `outerVariable` even though `outerFunction` has finished executing.

- When `closureFunction` is called, it logs the value of `outerVariable`, demonstrating that it still has access to the variables of its outer scope.

Closures are useful for creating private variables, maintaining state across function calls, and implementing various design patterns in JavaScript. They are a fundamental concept in JavaScript and play a crucial role in functional programming and creating modular and maintainable code.

NOTE : We can use the closure similar to classes. We can return an object where the inner function will called and the outer function variables will remain in constant state.

Modules :

ES6 (ECMAScript 2015) introduced the concept of modules to JavaScript. Modules provide a way to organize code into reusable units and encapsulate the implementation details. The module system helps in creating a more maintainable and scalable codebase. Here is a basic overview of ES6 modules:

Exporting from a Module:

To export variables, functions, or classes from a module, you use the `export` keyword:

Example :

```
// myModule.js
    export const myVariable = 42;
    export function myFunction() {
      // function implementation
    }
    export class MyClass {
      // class implementation
    }
```

Importing in Another Module:

To use the exported values in another module, you use the `import` statement:

Example :

```
// anotherModule.js

    import { myVariable, myFunction, MyClass } from
    './myModule';

    console.log(myVariable);

    myFunction();

    const instance = new MyClass();
```

Default Exports:

You can also have a default export in a module. Only one default export is allowed per module.

Example :

```
// myModule.js

    const myVariable = 42;

    export default myVariable;

// anotherModule.js

    import myVariable from './myModule';

    console.log(myVariable);
```

Combining Named and Default Exports:

You can use a combination of named exports and a default export in the same module.

Example :

```
// myModule.js

    export const constantValue = 42;

    export default function() {
```

```
    // default function implementation
}
// anotherModule.js
import defaultFunction, { constantValue } from './myModule';
console.log(constantValue);
defaultFunction();
```

Module Syntax in HTML:

When using ES6 modules in the browser, you can use the `type="module"` attribute in the script tag:

HTML Example :

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>ES6 Modules</title>
</head>
<body>
  <script type="module" src="app.js"></script>
</body>
</html>
```

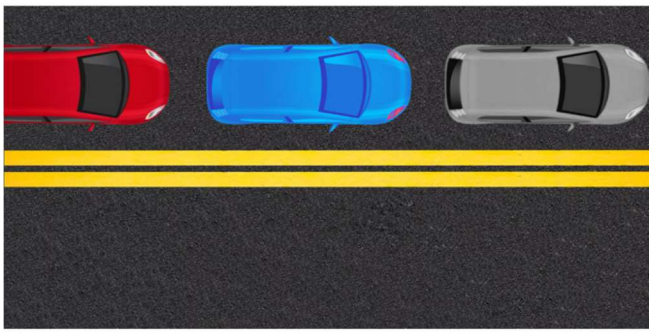
js Example :

```
// app.js
```

```
import { myFunction } from './myModule';
myFunction();
```

Synchronous:

In a sequential and blocking manner. In a synchronous process, each operation must complete before the next one starts. This means that if there is a delay in any operation, it will block the execution of subsequent operations until the current one finishes.



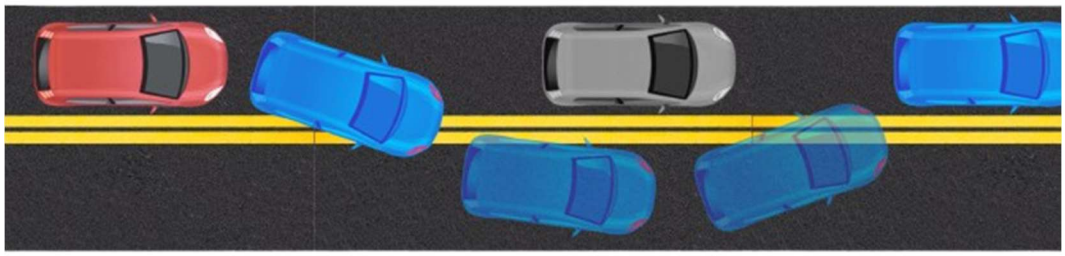
Example (in pseudocode):

```
console.log("Step 1");
console.log("Step 2");
console.log("Step 3");
// Output: Step 1
//         Step 2
//         Step 3
```

In this synchronous example, each step is executed in order, one after the other.

Asynchronous:

Allow multiple operations to overlap in time. In an asynchronous process, a task can start without waiting for the previous one to complete. Asynchronous operations are commonly used in situations where there may be delays, such as fetching data from a remote server or reading a file.



Example (in pseudocode using JavaScript):

```
console.log("Step 1");
// Assume fetchData is an asynchronous function that takes a callback
fetchData(function(data) {
  console.log("Step 2: " + data);
});
console.log("Step 3");
// Output: Step 1
//         Step 3
//         Step 2: [data]
```

In this asynchronous example, "Step 1" and "Step 3" are executed immediately, and "Step 2" is executed later when the asynchronous `fetchData` operation completes.

Asynchronous operations are often used to improve the efficiency and responsiveness of programs, especially in scenarios where waiting for a task to complete would result in unnecessary delays.

In modern JavaScript, asynchronous operations are commonly handled using features like Promises, async/await, and callback functions. These mechanisms allow developers to work with asynchronous code in a more organized and readable manner.

Error Handling :

Error handling is a critical aspect of writing robust and reliable software. Properly managing errors ensures that your program can gracefully handle unexpected situations, recover from failures, and provide meaningful feedback to users or developers. Here are some common approaches to error handling in programming:

1. **Try-Catch Blocks:*

In languages like JavaScript, Java, and Python, you can use `try-catch` blocks to handle exceptions (errors). Code within the `try` block is executed, and if an exception occurs, the corresponding `catch` block is executed.

Example :

```
try {
  // Code that may throw an exception
} catch (error) {
  // Handle the exception
  console.error("An error occurred:", error.message);
}
```

2. **Error Objects:*

When an error occurs, it's often represented by an object that contains information about the error, such as an error message and a stack trace. This object can be caught and used for further analysis or logging.

Example :

```
try {
  // Code that may throw an exception
} catch (error) {
  if (error instanceof SomeSpecificError) {
    // Handle a specific type of error
  } else {
    // Handle other types of errors
    console.error("An error occurred:", error.message);
  }
}
```

3. **Throwing Errors:*

You can throw your own custom errors or exceptions using the `throw` keyword. This is useful for signaling specific issues in your code.

Example :

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

4. **Promises and Async/Await:*

When working with asynchronous code, error handling is often done using Promises and the `catch` method, or with `async/await` syntax.

Example :

```
someAsyncFunction()
  .then(result => {
    // Handle success
  })
  .catch(error => {
    // Handle errors
    console.error("An error occurred:", error.message);
  });
//javascript
async function someAsyncFunction() {
  try {
    const result = await fetchSomething();
    // Handle success
  } catch (error) {
    // Handle errors
    console.error("An error occurred:", error.message);
  }
}
```

5. **Logging:*

Logging errors is crucial for troubleshooting and debugging. Log error details, including the error message, stack trace, and any relevant context, to help identify and fix issues.

Example :

```
try {
  // Code that may throw an exception
} catch (error) {
  // Log the error
  console.error("An error occurred:", error);
}
```

6. **Graceful Degradation:*

In situations where an error won't cause a critical failure, you may choose to gracefully degrade the functionality or provide a fallback mechanism.

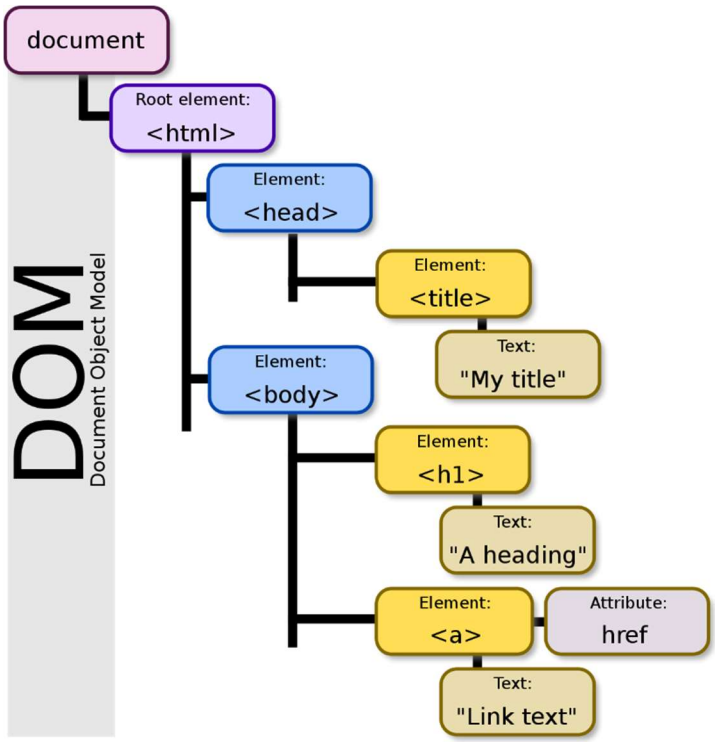
Example :

```
try {
  // Code that may throw an exception
} catch (error) {
  // Use a fallback or provide a degraded functionality
  console.warn("An error occurred, using fallback:", error.message);
}
```

DOM :

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of a document as a tree of objects, where each object corresponds to a part of the document, such as elements, attributes, text content, etc. The DOM provides a way for programs to manipulate the structure, style, and content of web documents.

Here's a detailed explanation of key concepts related to the DOM:



1. **Document Structure:**

- **Document:** The top-level object representing the entire HTML or XML document.

- **Element:** Represents HTML or XML elements in the document (e.g., `<div>`, `<p>`, `<a>`).

- **Attribute:** Represents the attributes of HTML or XML elements (e.g., `class`, `id`, `src`).

2. **Tree Structure:**

- The DOM is organized as a tree structure where each node in the tree represents an object in the document.

- The top of the tree is the `document` object, and each HTML element, attribute, and text content is represented as a node in the tree.

3. **Node Types:**

- **Element Node:** Represents an HTML or XML element.

- **Attribute Node:** Represents an attribute of an element.

- **Text Node:** Represents the text content within an element.

- ***Comment Node:**** Represents comments in the HTML code.

*4. **Traversal:***

- The DOM allows you to traverse and navigate the tree using methods and properties.

- Common traversal methods include `getElementById`, `getElementsByTagName`, `getElementsByClassName`, and others.

*5. **Manipulation:***

- The DOM provides methods to manipulate the content and structure of the document.

- You can create, delete, and modify elements and attributes using methods like `createElement`, `appendChild`, `removeChild`, `setAttribute`, etc.

*6. **Events:***

- The DOM allows you to attach event handlers to elements to respond to user interactions or other events (e.g., click, hover, submit).

- Event handling is an essential part of creating interactive web pages.

*7. **Style and Layout:***

- The DOM also exposes properties to manipulate styles and layout, allowing dynamic changes to the appearance of elements.

- CSS styles can be manipulated using the `style` property of DOM elements.

*8. **Browser Interaction:***

- The DOM is a platform-independent interface that allows scripting languages (usually JavaScript) to interact with web browsers.

- JavaScript is the most common language used to manipulate the DOM in web development.

Example (JavaScript):

```
// Accessing an element by its ID
const myElement = document.getElementById('myId');
```

```
// Modifying the content of the element
myElement.textContent = 'Hello, DOM!';
// Creating a new element and appending it to the document
const newElement = document.createElement('p');
newElement.textContent = 'A new paragraph';
document.body.appendChild(newElement);
```

Element Selectors :

Element selectors are used to target the tags or content in a document. We use the below Element selectors in js to select one or more html elements at a time.

1. getElementsById() > returns a Null if not found or Element

2. getElementsClassName() > returns the HTML collection

3. getElementsByTagName() > returns the HTML Collection based upon the tagName inside the document.

4. querySelector() > Return 1st element or Null if nothing

5. querySelectorAll() > Return Node list

`getElementById()` Method:

This method is used to select a single HTML element with a specified ID attribute. It returns the element as an object, or `null` if no element with the specified ID is found.

Example:

```
// HTML: <div id="myDiv">Hello, World!</div>
// JavaScript
const myDiv = document.getElementById('myDiv');
if (myDiv) {
    console.log(myDiv.textContent); // Outputs: Hello, World!
} else {
```

```
    console.log('Element with ID "myDiv" not found.');
}
```

In this example, `getElementById('myDiv')` is used to select the `<div>` element with the ID "myDiv" in the HTML document.

It seems there's another small typo in your request. The correct method is `getElementsByClassName()` (not `getElementsClassName()` with an "s"). The correct method is plural and is used to select multiple HTML elements based on their class names. Here's the correct usage:

`getElementsByClassName()` Method:

This method is used to select multiple HTML elements with a specified class name. It returns a live HTMLCollection, which is similar to an array, containing all elements that have the specified class.

Example:

```
<!-- HTML: -->
<div class="box">Box 1</div>
<div class="box">Box 2</div>
<div class="box">Box 3</div>
// JavaScript:
const boxes = document.getElementsByClassName('box');
// Accessing elements in the HTMLCollection:
for (let i = 0; i < boxes.length; i++) {
    console.log(boxes[i].textContent);
}
```

In this example, `getElementsByClassName('box')` is used to select all elements with the class "box" in the HTML document. The resulting HTMLCollection (`boxes`) can be iterated through to access and manipulate each element.

Note: The `getElementsByClassName` method returns a live collection, meaning if elements with the specified class are added or removed after the collection is obtained, the collection automatically updates to reflect the changes.

`getElementsByTagName()` Method:

This method is used to select multiple HTML elements with a specified tag name. It returns a live HTMLCollection, similar to an array, containing all elements that have the specified tag.

Example:

```
<!-- HTML: -->
<p>Paragraph 1</p>
<p>Paragraph 2</p>
<p>Paragraph 3</p>
// JavaScript:
const paragraphs = document.getElementsByTagName('p');
// Accessing elements in the HTMLCollection:
for (let i = 0; i < paragraphs.length; i++) {
    console.log(paragraphs[i].textContent);
}
```

In this example, `getElementsByTagName('p')` is used to select all `<p>` elements in the HTML document. The resulting HTMLCollection (`paragraphs`) can be iterated through to access and manipulate each paragraph element.

Note: Similar to `getElementsByClassName()`, the `getElementsByTagName` method returns a live collection, which means it automatically updates if elements with the specified tag name are added or removed after the collection is obtained.

`querySelector()` Method:

The `querySelector` method is a powerful and versatile method in JavaScript that allows you to select a single HTML element using a CSS selector. It returns the

first element that matches the specified selector, or `null` if no matching element is found.

This method takes a CSS selector as an argument and returns the first element in the document that matches the selector.

Example:

```
<!-- HTML: -->

<div id="myDiv">Hello, World!</div>

<p class="paragraph">This is a paragraph.</p>

// JavaScript:

const myDiv = document.querySelector('#myDiv'); // Selects element with ID 'myDiv'

const firstParagraph = document.querySelector('.paragraph'); // Selects first element with class 'paragraph'

console.log(myDiv.textContent); // Outputs: Hello, World!

console.log(firstParagraph.textContent); // Outputs: This is a paragraph.
```

In this example, `querySelector('#myDiv')` selects the element with the ID 'myDiv', and `querySelector('.paragraph')` selects the first element with the class 'paragraph'.

Additional Notes:

- The `querySelector` method allows you to use any valid CSS selector, providing great flexibility.

- If you want to select multiple elements that match a selector, you can use `querySelectorAll()`.

- It's a good practice to check if the selected element is not `null` before attempting to access its properties or call its methods.

Example:

```
const element = document.querySelector('.someClass');

if (element) {
```

```
 // Do something with the element
 console.log(element.textContent);
} else {
 console.log('No element found with the specified selector.');
}
```

This approach helps prevent errors when dealing with non-existent elements.

DOM Navigation :

DOM (Document Object Model) navigation involves moving around and accessing different parts of the document tree. In JavaScript, you can navigate the DOM using various properties and methods provided by the DOM API. Here are some common methods for DOM navigation:

*1. **Parent Node:***

- The `parentNode` property is used to get the parent node of an element. We can also use the parentElement property as well

Syntax :

```
const childElement = document.getElementById('childElementId');
const parentElement = childElement.parentNode;
const parent = childElement.parentElement;
```

Example :

```
const parent = p1.parentNode;
const parent = p1.parentElement;
```

*2. **Child Nodes:***

- The `childNodes` property returns a NodeList of child nodes of an element. Note that this includes text nodes, comments, and other types of nodes.

Syntax :

```
const parentElement = document.getElementById('parentElementId');
```

const childNodes = parentElement.childNodes;

Example :

```
const pList = div.children;

const pList = div.childNodes;
const pList = div.childElementCount; // returns the children count
const pList = div.hasChildNodes(); // returns the true or false
const pList = div.firstElementChild;
```

3. **First and Last Child:**

- The `firstChild` and `lastChild` properties return the first and last child nodes of an element, respectively.

***Syntax* :**

const parentElement = document.getElementById('parentElementId');

const firstChild = parentElement.firstChild;

const lastChild = parentElement.lastChild;

Example :

```
const pList = div.lastChild; // returns the last child
const pList = div.firstChild;
```

4. **Next and Previous Sibling:**

- The `nextSibling` and `previousSibling` properties return the next and previous sibling nodes of an element, respectively.

Example :

const siblingElement = document.getElementById('siblingElementId');

const nextSibling = siblingElement.nextSibling;

const previousSibling = siblingElement.previousSibling;

5. **Next and Previous Element Sibling:**

- The `nextElementSibling` and `previousElementSibling` properties return the next and previous element sibling nodes, excluding non-element nodes (such as text nodes).

Example :

const siblingElement = document.getElementById('siblingElementId');

const nextElementSibling = siblingElement.nextElementSibling;

const previousElementSibling = siblingElement.previousElementSibling;

6. ***Element Children:***

- The `children` property returns a collection of element nodes that are direct children of the specified element.

Example :

const parentElement = document.getElementById('parentElementId');

const elementChildren = parentElement.children;

Adding and Changing Attributes :

Adding Elements:

*1. **Create a New Element:***

// Create a new paragraph element

Syntax :

const newParagraph = document.createElement(tagName);

newParagraph.textContent = ‘textContent';

Example :

```
const p1 = document.createElement('p');
p1.textContent = 'para1';
```

*2. **Append the Element to the DOM:***

// Append the new paragraph to the body of the document

Syntax :

document.body.appendChild(newElement);

Example :

```
div.appendChild(p1);
```

3. **Prepend the Elements to the DOM : *

Used to insert the element at the first of the child elements.

Syntax :

Parent.prepend(childNode);

Example :

```
div.prepend(p3);
```

4. **InsertBefore : *

Insert the element before the selected element.

Syntax :

Parent.insertBefore(insertElement, referenceElement);

Example :

```
div.insertBefore(p4, div.lastChild);
```

Changing Elements:

*1. **Access an Existing Element:***

// Access an existing element by its ID

const existingElement = document.getElementById('existingElementId');

*2. **Change Text Content:***

// Change the text content of an existing element

existingElement.textContent = 'Updated text content';

*3. **Change Attribute Value:***

// Change the value of the 'src' attribute of an image

const imageElement = document.getElementById('myImage');

```
imageElement.setAttribute('src', 'newImagePath.jpg');
```

*4. **Change CSS Style:***

```
// Change the background color of an existing element
existingElement.style.backgroundColor = 'lightblue';
```

We can create a complete HTML structure using the js but we need to have the basic structure of html page. For that we need to follow this procedure.

1. Create a element using "createElement"

2. Edit the element using the "textContent" and also provide Id by using the ".id", for class using ".class"

3. Then append or prepend the element to the HTML document using the "append" or "prepend". You can even select where you want to insert using the "insertBefore"

Example :

```
document.body.insertBefore( createdTag, Target);
```

4. To remove a created element or existing element, then use the remove child.

Example :

```
document.body.removeChild(target);
```

Event Listeners :

In JavaScript, event listeners are used to detect and respond to events that occur in the browser, such as user interactions (e.g., clicks, keypresses) or changes in the DOM (Document Object Model). Event listeners are commonly used to add interactivity to web pages. Here's a basic overview of how event listeners work:

1. **Event Types:

- Common events include "click," "mouseover," "keydown," "submit," etc.
- Choose the appropriate event type based on the action you want to capture.

2. **Event Target:*

- The event target is the DOM element on which the event occurs.

- It could be the entire document, a specific HTML element, or any other valid DOM element.

3. **Event Listener:*

- An event listener is a function that is called whenever a specified event type occurs on a specified target.

- You attach an event listener to an element using the `addEventListener` method.

Syntax:*

target.addEventListener(type, listenerFunction);

- *`target`: The DOM element to which the event listener is attached.*

- *`type`: A string representing the event type (e.g., "click," "keydown").*

- *`listenerFunction`: The function that will be called when the event occurs.*

Example:

```
// HTML:

<button id="myButton">Click me</button>
```

In this example, the `handleClick` function will be called when the button with the id "myButton" is clicked.

6. **Removing Event Listeners:*

- You can remove an event listener using the `removeEventListener` method.

- It's important to use the same function reference that was used during the `addEventListener` call.

Example :

```
button.removeEventListener('click', handleClick);
```

This removes the previously added click event listener.

NOTE : If we using an arrow function as an event listener and you need to remove it later, you face a challenge because arrow functions are anonymous and don't have a name. In such cases, you typically can't directly remove the arrow function using *removeEventListener()* because you don't have a reference to it.

*7. **Event Object:***

- The event listener function typically receives an event object as a parameter.

- This object contains information about the event, such as the type, target, and additional data depending on the event type.

Example :

```
function handleClick(event) {
    console.log('Button clicked!', event);
}
```

Event object will have a method called matches which is used to match the selectors like id, class and tags.

Example :

```
event.target.matches('#parent')
```

NOTE : We can add n - no.of event listeners in JavaScript

Example in js:

Onclick event :

Note : If when we trigger an child event then it's parent element event will also get triggered along with it. This is process called Bubbling phase.

Tip : To stop the bubble event we can use the stopPropagation() method.

EventListner will accept the 3 arguments.

1. **Events argument** – provides the events
2. **Callback function** – will mention the function that needs to run
3. **Capture** – will mention true if we want the parent to invoke 1st

Note : If you want to stop the execution once then we use the once as true in 3rd argument in eventListeners.

EventListeners and Types :

1. mouseover : *will add the effect when we hover on the target*

2. mouseout : *will add the effects when we hover out from the target*

3. mousedown : *Will add the effects when we press the mouse buttons*

4. mouseup : *will add the effects when we release the mouse button*

5. click : *will change the effects when we click on it*

6. Keydown : *will activate when we press a key on keyboard*

7. Keyup : *will activate when we release a key on keyboard*

8. Scroll : *Willl trigger when we scroll on the window*

Node List :

A node list is an array-like collection of nodes. Nodes typically represent elements in the Document Object Model (DOM), which is a hierarchical representation of the structure of an HTML or XML document. Node lists are commonly used when selecting multiple elements from the DOM using methods like querySelectorAll or properties like childNodes.

NOTE : Node lists will not have any built in methods like map and filter or reduce. But will have foreach loop

Note : Node list will not automatically reflect the changes for that we need to apply the changes seperately as these are static collection.

Example :

```
//HTML code
```

```
<button class = "Nodes"> button 1</button>
<button class = "Nodes"> button 2</button>
<button class = "Nodes"> button 3</button>
<button class = "Nodes"> button 4</button>
```

```
// Javascript code
let buttons = document.querySelectorAll('Nodes'); // create a static collection of array
let newButton = document.createElement('button'); // creating a button
newButton.textContent = "button 5"; // adding text to the button
newButton.classList = "Nodes"; // providing class Name to the button
document.body.append(newButton); // appending the button to the body
buttons = document.querySelectorAll('Nodes'); // updating the static collection of array
```

Note : We can use the remove method to remove the element from the collection. Even though we removed the elements from the body the node list will have the data for that we need to update once again.

Class List :

In JavaScript, the `classList` property is used to interact with the classes of an HTML element. It provides a convenient way to add, remove, toggle, and check for the presence of CSS classes on an element. The `classList` property is available on all elements in the DOM.

Here are some common methods and properties of the `classList` object:

1. **`add(className1, className2, ...)` method:*

Adds one or more class names to the element. If the class already exists, it will not be duplicated.

Example :

```
var element = document.getElementById('myElement');
element.classList.add('newClass', 'anotherClass');
```

2. **`remove(className1, className2, ...)` method:*

Removes one or more class names from the element.

Example :

```
var element = document.getElementById('myElement');
element.classList.remove('newClass', 'anotherClass');
```

3. **`toggle(className, force)` method:*

Toggles the presence of a class. If the class exists, it is removed; if it does not exist, it is added. The optional `force` parameter is a boolean that forces the class to be added or removed based on its truthiness.

Example :

```
var element = document.getElementById('myElement');
element.classList.toggle('active');
```

4. **`contains(className)` method:*

Checks if the element has a specific class. Returns `true` if the class exists, and `false` otherwise.

Example :

```
var element = document.getElementById('myElement');
if (element.classList.contains('active')) {
   // Do something
}
```

5. **`item(index)` method:*

Returns the class name at the specified index. This is useful if you want to access a specific class by its position in the list.

Example :

var element = document.getElementById('myElement');

var classNameAtIndex = element.classList.item(0);

6. **`length` property:*

Returns the number of classes in the element's `classList`.

Example :

var element = document.getElementById('myElement');

var numberOfClasses = element.classList.length;

Using the `classList` methods, you can easily manipulate the classes of HTML elements, which is commonly done in dynamic web applications to apply or remove styles based on user interactions or other events.

Call Back Hell :

"Callback hell" in JavaScript refers to a situation where multiple nested callbacks make the code difficult to read, understand, and maintain. This usually occurs when dealing with asynchronous operations, such as making multiple API calls or handling events.

Example :

```
function data1(callback){
    setTimeout(() => {
        console.log("data1 found");
        callback();
    }, 2000);
}

function data2(callback){
    setTimeout(() => {
        console.log("data2 found");
        callback();
    }, 3000);
}

function data3(callback){
        console.log("data3 found");
```

```
        callback();
}

data1(
    () => {
        data2(
            () => {
                data3(
                    () => {
                        console.log("all data found");
                    }
                );
            }
        );
    }
);
```

Note : Call backs are complicated as it is having a nested callbacks. To make the asynchronus functions more simplier we are having the promises and async/ await.

Promises :

Promises are a feature in JavaScript that were introduced to handle asynchronous operations more elegantly. They provide a cleaner and more readable syntax for dealing with asynchronous code compared to traditional callback-based approaches. Promises represent a value which might be available now, or in the future, or never.

A Promise can be in one of three states:

*1. **Pending:** The initial state; the promise is neither fulfilled nor rejected.*

*2. **Fulfilled:** The operation completed successfully, and the promise has a resulting value.*

*3. **Rejected:** The operation failed, and the promise has a reason for the failure.*

Example :

```
function promiseExample(){
    return new Promise((resolve, reject) => {
        const found = true;
        setTimeout(() => {
            if (found) {
```

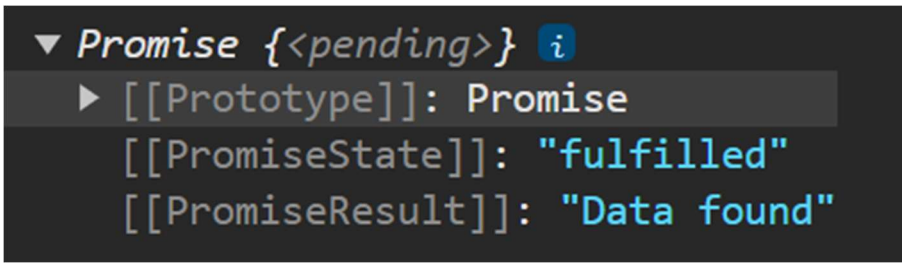
```
                resolve("Data found");
            } else {
                reject("data not found");
            }
        }, 1000);
    });
}

console.log(promiseExample());
```

Handling promises using the method chaining for handling multiple promises

Example :

```
      function p1(){

    return new Promise((resolve, reject) => {
        const found = true;
        setTimeout(() => {
            if (found) {
                resolve("Data found");
            } else {
                reject("data not found");
            }
        }, 1000);
    });
}

function p2(response){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (response == 'Data found') {
                resolve("Data found");
            } else {
                reject("data not found");
            }
        }, 1000);
    });
}

const result = p1().then((response) => {
        return p2(response);
});

console.log(result);
```

Output :



- The `Promise` constructor takes a function with two arguments: `resolve` and `reject`. These are functions provided by the JavaScript runtime to signal the completion or failure of the asynchronous operation.

- Inside the function, you perform your asynchronous operation. If it succeeds, you call `resolve` with the result. If it fails, you call `reject` with an error message.

- The `then` method is used to handle the case when the Promise is fulfilled, and the `catch` method is used to handle the case when the Promise is rejected.

Promises Methods :

1. Promise.all() :

Can use the array of promises to make them synchronous code and will return the resolves into an array, but if any promise failed to resolve then it returns the reject message.

Example :

```
let p1 = new Promise((resolve, reject) =>{
   resolve("p1 completed successfully");
})

let p2 = new Promise((resolve, reject) =>{
    resolve("p2 completed successfully");
 })
 let p3 = new Promise((resolve, reject) =>{
    reject("p3 failed");
 })
```

```
Promise.all([p1, p2, p3]).then( message =>
console.log(message)).catch(err =>      console.error(err));
```

//output : P3 failed

2. Promise.race() :

Will act similar to all() but the only difference is it will only return a single value.

Example :

```
let p1 = new Promise((resolve, reject) =>{
   resolve("p1 completed successfully");
})
let p2 = new Promise((resolve, reject) =>{
   resolve("p2 completed successfully");
})
let p3 = new Promise((resolve, reject) =>{
   reject("p3 completed successfully");
})
Promise.race([p1, p2, p3]).then( message =>
console.log(message)).catch(err => console.error(err));
//output : p1 completed successfully
```

Promises help avoid callback hell and make it easier to reason about asynchronous code. They are also the foundation for the `async/await` syntax, which provides an even more concise way to work with asynchronous code in modern JavaScript.

Async and Await :

`async` and `await` are keywords in JavaScript that are used in conjunction with Promises to write asynchronous code in a more readable and synchronous-looking style.

1. **`async` Function:*

The `async` keyword is used to declare an asynchronous function. An asynchronous function always returns a Promise. It allows the use of the `await` keyword inside it.

Example :

```
async function myAsyncFunction() {
  // Code here
}
```

2. **`await` Operator:*

The `await` keyword is used to pause the execution of an `async` function until the Promise is settled (resolved or rejected). It can only be used inside an `async` function.

Example :

```
async function fetchData() {
  let result = await someAsyncOperation();
  console.log(result);
}
```

In this example, `fetchData` is an asynchronous function that calls `someAsyncOperation`. The `await` keyword pauses the execution of `fetchData` until `someAsyncOperation` is resolved, and then it continues with the result.

Here's an example using `async` and `await` with the `fetch` function, which is commonly used for making asynchronous HTTP requests in modern JavaScript:

Example :

```
function task1(){
```

```
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('task1');
        }, 1000);
    })
}

function task2(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject('task2');
        }, 2000);
    })
}

async function main(){
    try{
        const result = await task1();
        console.log(result);
        const result2 = await task2();
        console.log(result2);
    }catch(e){
        console.error(e);
    }
}
```

```
main();
```

Note : Using `async` and `await` makes asynchronous code more readable and easier to reason about compared to traditional callback-based or promise-chaining approaches.

JSON Files :

JSON (JavaScript Object Notation) files are a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. They are primarily used to transmit data between a server and a web application as an alternative to XML.

JSON files consist of key-value pairs and arrays, organized into a hierarchical structure. The keys are strings, and the values can be strings, numbers, booleans, arrays, objects, or null.

Example :

```
{
 "name": "John Doe",
 "age": 30,
 "isStudent": false,
 "address": {
  "street": "123 Main St",
  "city": "Anytown",
  "country": "USA"
 },
      "friends": ["Alice", "Bob", "Charlie"]
}
```

In this example:

- `"name"`, `"age"`, and `"isStudent"` are key-value pairs with string keys and various types of values.

- `"address"` is a nested object containing its own key-value pairs.

- `"friends"` is an array containing strings as elements.

JSON files are commonly used for configuration files, data storage, and data exchange between a client and a server in web development. They are also used in many other contexts, such as APIs, logging, and data serialization. JSON has become ubiquitous in web development due to its simplicity, flexibility, and ease of use.

Note : To create JSON files we use the '.json' extension.

Using JSON :

In javascript we use the built-in methods in order to manipulate the JSON.

1. JSON.parse() : parsing JSON strings into JavaScript objects

2. JSON.stringify() : converting JavaScript objects into JSON strings

Example :

```
let jsonStr = '{"Name": "test", "description": "test description"}';
let jsObj = JSON.parse(jsonStr);
console.log(jsObj); //{Name: 'test', description: 'test description'}
let jsonConvert = JSON.stringify(jsObj);
console.log(jsonConvert); // {"Name":"test","description":"test description"}
```

Fetching Data using API's :

To fetch the data using the API's we use the fetch() function, which is a HTTP Requests to fetch resources. fetch() is a built-in method.

When you are using the fetch() it will create a object for the request() class in the background and it will accept the url and the options especially like methods : GET, POST

NOTE : By default javascript will take the GET method when you are using the default fetch

Syntax :

fetch(url, {options})

Using `fetch()`:

Example :

```
fetch(url)
    .then( (response) => response.json())
    .then( (data) => console.log(data.activity) )
    .catch( error => console.log(error) );
```

It will not throw any error if there is an issue. For that we can use the ok object where it uses the true/ false for that we can write the code as below.

Example :

```
// Will throw if there is an error in API
    fetch("https://pokeapi.co/api/v2/pokemon/ditto")
      .then(res => {
        if(res.ok){
          return res.json();
        }
        else{
          throw new Error("Cannot be found");
        }
      } )
      .then(data => console.log(data))
      .catch(error => console.error(error));
```

How will the fetch works ?

1. Send a request to the API endpoint.
2. Check if the response is successful.
3. Parse the response data (if successful).
4. Handle errors if they occur.

Using promise and resolve :

```
function getData(){

    return fetch('https://jsonplaceholder.typicode.com/comments/1').
             catch(error => console.log(error));
}

let data = getData();
data.then(response => response.json()).
      then(data => console.log(data));
```

Using async and await function :

```
async function getData(){

    let response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
    let data = await response.json();
    console.log(data);
}
getData();
```

Example :

```
FetchData();
async function FetchData(){
   try{
      const res = await fetch("https://pokeapi.co/api/v2/pokemon/charizard");
      if(res.ok){
         const data = await res.json();
         console.log(data);
      }
```

```
            else{
                throw new Error("Cannot fetchPokemon ");
            }
        }
        catch(err){
            console.log(err);
        }
    }
```

HTTP response codes :

HTTP response codes are standardized status codes returned by a web server in response to a client's request made to the server. These codes indicate the outcome of the request and provide information about whether the request was successful, encountered an error, or requires further action from the client.

- **1xx Informational:**

These are provisional responses indicating that the server has received the request and is processing it.

*- ****100** Continue**: The client can continue with its request.*

*- ****101** Switching Protocols**: The server is changing protocols according to the client's request.*

- **2xx Success:**

These indicate that the request was received, understood, and processed successfully.

*- ****200** OK**: The request was successful.*

*- ****201** Created**: The request resulted in a new resource being created.*

*- ****204** No Content**: The server successfully processed the request but is not returning any content.*

- **3xx Redirection:**

These indicate that further action needs to be taken by the client to complete the request.

*- ****301** Moved Permanently**: The requested resource has been permanently moved to a new location.*

*- ****304** Not Modified**: The resource has not been modified since the last request.*

- **4xx Client Error**:

These indicate that there was an error on the client's side, such as invalid request syntax or unauthorized access.

*- ****400** Bad Request**: The request could not be understood by the server due to malformed syntax.*

*- ****403** Forbidden**: The client does not have permission to access the requested resource.*

*- ****404** Not Found**: The requested resource could not be found on the server.*

- **5xx Server Error:**

These indicate that there was an error on the server's side while processing the request.

*- ****500** Internal Server Error**: A generic error message indicating that something went wrong on the server.*

*- ****503** Service Unavailable**: The server is currently unavailable due to overload or maintenance.*

Hall of Fame properties in javaScript :

1. checked :

In the context of JavaScript and HTML, the `.checked` property is commonly associated with form elements, particularly checkboxes. It is used to determine or set the checked state of a checkbox input element.

Reading the Checked State:

```
// Assuming you have an HTML checkbox with the id "myCheckbox"
let checkboxElement = document.getElementById("myCheckbox");
// Checking if the checkbox is checked
if (checkboxElement.checked) {
    console.log("Checkbox is checked");
} else {
    console.log("Checkbox is not checked");
}
```

Setting the Checked State:

You can also use the `.checked` property to programmatically set the checked state of a checkbox.

Example in javaScript

```
// Assuming you have an HTML checkbox with the id "myCheckbox"
let checkboxElement = document.getElementById("myCheckbox");
// Setting the checkbox to be checked
checkboxElement.checked = true;
```

This is often used when you want to manipulate the state of checkboxes dynamically through JavaScript.

Example HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Checkbox Example</title>
</head>
<body>
    <label for="myCheckbox">Check me:</label>
    <input type="checkbox" id="myCheckbox">
    <script>
        // Your JavaScript code here
    </script>
</body>
</html>
```

This :

In JavaScript, the `this` keyword is a reference to the current execution context, and its value depends on how a function is called. The behavior of `this` can be a common source of confusion for developers, as it varies in different situations. Here are the main rules for determining the value of `this`:

1. **Global Context:*

- In the global scope (outside any function), `this` refers to the global object. In a browser environment, the global object is usually `window`.

Example :

```
console.log(this); // Output: [object Window] (in a browser environment)
```

2. **Function Context:*

- Inside a function, the value of `this` depends on how the function is invoked.

- In a regular function (not an arrow function), `this` refers to the object that the function is a method of or the global object if the function is not a method of any object.

Example :

```
function showThis() {
  console.log(this);
}
showThis(); // Output: [object Window] (in a browser environment)
```

3. **Method Context:*

- When a function is a method of an object, `this` refers to the object the method is called on.

Example :

```
const myObject = {
  myMethod: function() {
    console.log(this);
  }
};
myObject.myMethod(); // Output: [object Object] (referring to myObject)
```

4. **Constructor Context:*

- When a function is used as a constructor with the `new` keyword, `this` refers to the newly created object.

Example :

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

```
console.log(john.name); // Output: John
```

5. **Event Handler Context:*

- In event handler functions, `this` often refers to the DOM element that triggered the event.

Example :

```
document.getElementById('myButton').addEventListener('click', function()
{
  console.log(this); // Output: [object HTMLButtonElement]
});
```

6. **Arrow Functions:*

- Arrow functions do not have their own `this`. Instead, they inherit `this` from the enclosing scope.

Example :

```
const myFunction = () => {
  console.log(this);
};
myFunction(); // Output: [object Window] (in a browser environment)
```

It's important to be aware of the context in which a function is called to understand the value of `this`. Arrow functions can be particularly helpful in avoiding some common pitfalls related to `this` behavior.

Sort :

In JavaScript, the `sort` method is used to arrange the elements of an array. The default behavior of `sort` is to convert the elements into strings and then compare their sequences of UTF-16 code units. This can lead to unexpected results when sorting numbers. To sort numbers correctly, a compare function should be provided.

Example :

```
const numbers = [5, 2, 8, 1, 6];
// Sorting numbers in ascending order
numbers.sort((a, b) => a - b);
console.log(numbers); // Outputs: [1, 2, 5, 6, 8]
```

In this example, the `sort` method is called with a compare function `(a, b) => a - b`, which ensures that the array is sorted in ascending order. If you want to sort in descending order, you can use `b - a` in the compare function.

The expression a - b is commonly used as the compare function in the sort method when sorting an array of numbers. This specific expression determines the sorting order based on the numerical values of a and b. The result of a - b is:

1. Negative if a is less than b

2. Zero if a is equal to b

3. Positive if a is greater than b

For sorting an array of objects based on a specific property, you can modify the compare function accordingly:

Example :

```
const people = [
  { name: 'John', age: 30 },
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 35 }
];
// Sorting people based on age in ascending order
people.sort((a, b) => a.age - b.age);
console.log(people);
// Outputs: [{ name: 'Alice', age: 25 }, { name: 'John', age: 30 }, { name: 'Bob', age: 35 }]
```

Date Objects :

In JavaScript, the `Date` object is used to work with dates and times. It provides a way to represent and manipulate dates and times, allowing you to perform various operations, such as getting the current date, formatting dates, adding or subtracting time, and more.

1. **Creating a Date Object:**

You can create a `Date` object using the `new Date()` constructor. If no arguments are provided, it gives the current date and time.

Example :

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

2. **Parsing a Date String:**

You can create a `Date` object by parsing a date string.

Example :

```
const dateString = '2024-01-18T12:00:00';
const specificDate = new Date(dateString);
console.log(specificDate);
```

3. **Getting and Setting Components:**

The `Date` object has methods to get and set various components of a date, such as the year, month, day, hour, minute, second, and millisecond.

Example :

```
const today = new Date();
const year = today.getFullYear();
const month = today.getMonth(); // Months are zero-based (0 = January)
const day = today.getDate();
console.log(`Year: ${year}, Month: ${month + 1}, Day: ${day}`);
```

4. **Formatting Dates:**

You can format dates using methods like `toLocaleString()` or combining individual components.

Example :

```
const formattedDate = today.toLocaleString('en-US', { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' });

console.log(formattedDate);
```

5. **Adding/Subtracting Time:**

You can perform operations to add or subtract time from a `Date` object.

Example :

```
const futureDate = new Date();

futureDate.setDate(futureDate.getDate() + 7); // Adding 7 days

console.log(futureDate);
```

Note : When you want a custom date then follow the Date constructor, The date constructor will have the following order : year, month, day, hour, min, sec, ms)