DATE OBJECT:

In JavaScript, the Date object is used for working with dates and times. It allows you to create, manipulate, and format dates and times. The Date object is built into the JavaScript language and provides several methods and properties for working with dates and times.

```
const date = new Date();
```

The Date object in JavaScript comes with a variety of built-in methods to perform various operations on dates and times. Here are some of the commonly used built-in methods for the Date object:

getFullYear(): Returns the year (4 digits) of the date.

getMonth(): Returns the month (0-11) of the date.

getDate(): Returns the day of the month (1-31) of the date.

getDay(): Returns the day of the week (0-6) of the date (0 is Sunday, 6 is Saturday).

getHours(): Returns the hours (0-23) of the time.

getMinutes(): Returns the minutes (0-59) of the time.

getSeconds(): Returns the seconds (0-59) of the time.

getMilliseconds(): Returns the milliseconds (0-999) of the time.

getTime(): Returns the number of milliseconds since January 1, 1970 (Unix timestamp).

MATH OBJECT:

The Math object in JavaScript provides a collection of mathematical constants and functions for performing various mathematical operations. It is not a constructor like the Date object and does not have properties that you can modify. Instead, it's a static object that contains only methods and constants. Here are some of the commonly used methods and constants provided by the Math object in JavaScript:

Math.pow(x, y): Returns x raised to the power of y.

Math.sqrt(x): Returns the square root of x.

Math.cbrt(x): Returns the cube root of x.

Math.round(x): Returns the nearest integer to x.

Math.floor(x): Returns the largest integer less than or equal to x.

Math.ceil(x): Returns the smallest integer greater than or equal to x.

Math.trunc(x): Returns the integer part of x by removing the fractional part.

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

Math.random() * (max - min) + min: Generates a random number between min (inclusive) and max (exclusive).

Math.min(x, y, ...args): Returns the smallest of the provided values.

Math.max(x, y, ...args): Returns the largest of the provided values.

Synchronous Code:

In synchronous code execution, tasks are performed one at a time, in a sequential and blocking manner. Each task must complete before the next one can begin. If a task takes a long time to execute, it can block the entire program, making it unresponsive.

Asynchronous Code:

In asynchronous code execution, tasks are initiated and then allowed to run independently without waiting for their completion. This means that multiple tasks can be initiated and run concurrently. Asynchronous code allows a program to remain responsive even when performing tasks that take time to complete.

In JavaScript, setTimeout and setInterval are two functions that allow you to execute code at specified intervals. They are often used for creating timers and scheduling functions to run asynchronously.

setTimeout Function:

setTimeout is used to execute a specified function or code snippet after a specified delay (in milliseconds). It schedules a single execution of the specified function.

Syntax: setTimeout(callback, delay);

callback: A function to be executed after the delay.

delay: The time (in milliseconds) to wait before executing the callback.

EXAMPLE:

```
setTimeout(function () {
  console.log("This code will run after 1000 milliseconds (1 second).");
}, 1000);
```

setInterval Function:

setInterval is used to repeatedly execute a specified function or code snippet at a defined time interval. It keeps executing the function at regular intervals until it's canceled.

Syntax: setInterval(callback, delay);

callback: A function to be executed at each interval.

delay: The time (in milliseconds) between each execution of the callback.

Example:

```
const intervalId = setInterval(function () {
  console.log("This code will run every 2000 milliseconds (2 seconds).");
}, 2000);
```

setInterval returns a unique interval identifier that can be used to cancel the repeated function execution using clearInterval.

EXAMPLE:

```
const intervalId = setInterval(function () {
  console.log("This code will run indefinitely.");
}, 1000);
// Cancel the repeated function execution after 5 seconds
setTimeout(function () {
  clearInterval(intervalId);
}, 5000);
```

Window Object:

In JavaScript, the window object is a global object that represents the browser window or the global environment in a web browser. It serves as the top-level object in the browser's Document Object Model (DOM) hierarchy and provides access to various properties and methods that allow you to interact with the browser and control the web page.

this keyword in js:

In JavaScript, the this keyword is a special keyword that refers to the current context or object within which it is used. The behaviour of this depends on how and where it is used, and it can have different values in different contexts. Understanding how this behaves is crucial for working with JavaScript objects and functions.

Here are some common scenarios in which the this keyword behaves differently:

Global Context:

In the global scope (outside of any function or object), this refers to the global object, which in a browser environment is the window object.

Function Context:

Inside a regular function, the value of this depends on how the function is called.

In strict mode ("use strict";), if the function is called without any context (i.e., not as a method of an object or not using call() or apply()), this is undefined. But without strict mode it points towards the window object. But in arrow function it points towards the window object.

```
function foo() {
  console.log(this); // In non-strict mode: window, In strict mode: undefined
}
foo();
```

Method Context:

Inside a method of an object, this refers to the object itself.

Example:

```
const myObject = {
  prop: 'Hello',
  method: function () {
    console.log(this.prop); // 'Hello'
  },
};
myObject.method();
```

Constructor Context:

Inside a constructor function (a function used with the new keyword to create objects), this refers to the newly created object.

Example:

```
function Person(name) {
  this.name = name;
}
```

```
const person1 = new Person('Alice');
console.log(person1.name); // 'Alice'
```

Explicit Binding:

You can explicitly set the value of this using methods like call(), apply(), or bind().

EX:

```
function greet() {
  console.log(`Hello, ${this.name}`);
}
const person = { name: 'Alice' };
greet.call(person); // Explicitly bind `this` to the `person` object.
```

Rest Parameter (...) :

The rest parameter allows you to collect a variable number of function arguments into an array. It provides a way to work with an indefinite number of arguments as an array within a function.

Here's how the rest parameter works:

```
function sum(...numbers) {
  let result = 0;
  for (let number of numbers) {
    result += number;
  }
  return result;
}
console.log(sum(1, 2, 3, 4, 5)); // 15
```

In this example, the sum function takes any number of arguments, and the rest parameter ...numbers collects them into an array called numbers. You can then loop through this array to perform operations on the arguments.

Spread Operator (...):

The spread operator allows you to spread the elements of an iterable (e.g., an array) into another array, function argument, or object. It is used for unpacking values from an iterable, making it easier to work with arrays, objects, and other iterable data structures.

Here are some common use cases for the spread operator:

1. Spreading Arrays:

```
const arr1 = [1, 2, 3];

const arr2 = [...arr1, 4, 5];

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

2. Combining Arrays:

```
const arr1 = [1, 2];

const arr2 = [3, 4];

const combinedArray = [...arr1, ...arr2];

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

3. Copying Arrays:

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

const copyArray = [...originalArray];

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

console.log(originalArray === copyArray); // false (not the same reference)
```

4. Spreading into Function Arguments:

```
function multiply(x, y, z) {
  return x * y * z;
}

const numbers = [2, 3, 4];

const result = multiply(...numbers);

console.log(result); // 24
```

5. Spreading into Objects (Object Cloning and Merging):

```
const person = { name: 'Alice', age: 30 };

const updatedPerson = { ...person, age: 31 };

console.log(updatedPerson); // { name: 'Alice', age: 31
```

In this example, the spread operator is used to create a new object (updatedPerson) by copying all properties from the person object and overriding the age property.

Array Destructuring:

Array destructuring enables you to unpack values from an array and assign them to variables. It is particularly useful when working with functions that return arrays or when you want to access specific elements of an array easily.

Example:

```
const myArray = [1, 2, 3];

// Destructuring assignment

const [a, b, c] = myArray;

console.log(a); // 1

console.log(b); // 2

console.log(c); // 3
```

You can also use array destructuring with rest elements to collect remaining items into a new array:

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

const [first, second, ...rest] = numbers;

console.log(first); // 1

console.log(second); // 2

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

Object Destructuring:

Object destructuring allows you to extract values from objects and assign them to variables with the same property names. It's especially helpful when you want to work with objects in a more readable and concise way.

Here's how object destructuring works:

```
const myObject = { name: 'Alice', age: 30 };

// Destructuring assignment

const { name, age } = myObject;

console.log(name); // 'Alice'
```

console.log(age); // 30

You can also assign values to new variable names using object destructuring:

const person = { name: 'Bob', age: 25 };

// Destructuring assignment with new variable names

const { name: fullName, age: years } = person;

console.log(fullName); // 'Bob'

console.log(years); // 25

Object Methods in JS:

1. Object.freeze(obj):

The Object.freeze() method is used to freeze an object, making it immutable. Once an object is frozen, you cannot add, delete, or modify its properties or values.

Attempting to change a property's value or add/delete properties will have no effect, and JavaScript will not throw an error.

This method recursively freezes all nested objects and their properties.

2. Object.isFrozen(obj):

The Object.isFrozen() method checks if an object is frozen (immutable). It returns true if the object is frozen, otherwise false.

3. Object.seal(obj):

The Object.seal() method seals an object, making it non-extensible. While you can still modify the values of existing properties, you cannot add or delete properties.

Unlike Object.freeze(), sealing an object allows you to change the values of its existing properties, but you cannot add or remove properties.

4. Object.isSealed(obj):

The Object.isSealed() method checks if an object is sealed (non-extensible). It returns true if the object is sealed, otherwise false.

5.Object.assign():

The Object.assign() method is a built-in method in JavaScript that is used for copying the values of all enumerable properties from one or more source objects into a target object. It is often used to create a new object by merging properties from multiple source objects into a single destination (target) object. The Object.assign() method does a shallow copy of object properties, meaning that nested objects are not deeply copied; rather, references to nested objects are copied.

Here's the basic syntax of Object.assign():

Object.assign(target, source1, source2, ...);

target: The target object that will receive the copied properties. This object will be modified in place and also returned as the result.

source1, source2, ...: One or more source objects whose properties will be copied into the target object.

Here's an example of how to use Object.assign():

Example:

```
const target = { a: 1, b: 2 };

const source1 = { b: 3, c: 4 };

const source2 = { d: 5 };

// Merging properties from source1 and source2 into target

const mergedObject = Object.assign(target, source1, source2);

console.log(target); // { a: 1, b: 3, c: 4, d: 5 }

console.log(mergedObject === target); // true (the same object reference is returned).
```

In this example:

target is the object that receives the merged properties.

source1 and source2 are the source objects whose properties are copied into target.

The properties from source1 and source2 are merged into target, and the resulting modified target object is returned.

It's important to note that Object.assign() modifies the target object in place and also returns the modified target object. If there are properties with the same names in both the target and source objects, the properties in the source objects overwrite the corresponding properties in the target object. Additionally, if any of the source objects are null or undefined, they are simply ignored.

6. Object.keys(obj):

The Object.keys() method returns an array containing the keys of the given object.

7. Object.values(obj):

The Object.values() method returns an array containing the values of the enumerable properties of the given object.

8. Object.entries(obj):

The Object.entries() method returns an array of key-value pairs (entries) as arrays, where each sub-array contains a property name (key) and its associated value. It provides an easy way to iterate through both keys and values of an object.

structuredClone():

The primary purpose of structuredClone() is to create a deep clone of an object. It can clone not only simple objects but also complex data structures, such as nested objects, arrays, functions (in some cases), dates, and more.

EXAMPLE:

```
const originalObject = {
  name: 'Alice',
  age: 30,
  friends: ['Bob', 'Charlie'],
};
const clonedObject = structuredClone(originalObject);
console.log(clonedObject);
```

JSON:

JSON (JavaScript Object Notation) is a lightweight data interchange format used for storing and exchanging data between a server and a client, or between different parts of a JavaScript application. It is a text-based format that is easy for both humans and machines to read and write.

JSON Objects:

JSON objects are collections of key-value pairs. Keys are strings, and values can be strings, numbers, objects, arrays, booleans, null, or nested JSON objects.

```
let obj={
  "name": "Alice",
  "age": 30,
  "city": "New York"
}
```

1. JSON.stringify():

The JSON.stringify() method is used to convert a JavaScript object or value into a JSON-formatted string.

It takes an object or value as its argument and returns a string representing the JSON version of that object or value.

Example:

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York",
};
const jsonString = JSON.stringify(person);
console.log(jsonString);
```

2. JSON.parse():

The JSON.parse() method is used to parse a JSON-formatted string and convert it into a JavaScript object.

It takes a JSON-formatted string as its argument and returns a JavaScript object.

Example:

```
const jsonString = '{"name":"Alice","age":30,"city":"New York"}';
const person = JSON.parse(jsonString);
console.log(person.name); // "Alice"
```

Shallow Copy:

A shallow copy of an object creates a new object with a new reference, but it does not create new copies of nested objects or arrays within the original object. Instead, it copies references to those nested objects or arrays. As a result, changes made to nested objects in the copied object will affect the original object, and vice versa.

Here's how to achieve a shallow copy in JavaScript:

1.Using the spread operator (...):

```
const originalObject = { a: 1, b: { c: 2 } };
const shallowCopy = { ...originalObject };
```

```
// Modifying a property in the shallow copy
shallowCopy.b.c = 3;
console.log(originalObject.b.c); // 3 (originalObject is affected)
```

2.Using Object.assign():

```
const originalObject = { a: 1, b: { c: 2 } };
const shallowCopy = Object.assign({}, originalObject);
// Modifying a property in the shallow copy
shallowCopy.b.c = 3;
console.log(originalObject.b.c); // 3 (originalObject is affected)
```

Deep Copy:

A deep copy of an object creates a completely independent copy of the original object, including all nested objects and arrays. Changes made to the copied object or its nested objects will not affect the original object, and vice versa.

Here's an example of how to achieve a deep copy using a structuredClone():

```
// Create an object to be deep copied
const originalObject = {
  a: 1,
  b: {
    c: 2,
    d: [3, 4],
  },
};
// Deep copy using structuredClone()
const deepCopy = structuredClone(originalObject);
// Modify the deep copy
deepCopy.b.c = 99;
deepCopy.b.d.push(5);
// Verify that the originalObject is not affected
console.log(originalObject); // { a: 1, b: { c: 2, d: [3, 4] } }
console.log(deepCopy);       // { a: 1, b: { c: 99, d: [3, 4, 5] } }
```

2.Using JSON.parse() and JSON.stringify():

```
// Create an object to be deep copied
const originalObject = {
  a: 1,
  b: {
    c: 2,
    d: [3, 4],
  },
};
// Deep copy using JSON.stringify() and JSON.parse()
const deepCopy = JSON.parse(JSON.stringify(originalObject));
// Modify the deep copy
deepCopy.b.c = 99;
deepCopy.b.d.push(5);
// Verify that the originalObject is not affected
console.log(originalObject); // { a: 1, b: { c: 2, d: [3, 4] } }
console.log(deepCopy);       // { a: 1, b: { c: 99, d: [3, 4, 5] } }
```

In JavaScript, the call(), apply(), and bind() methods are used to manipulate the context (the value of the this keyword) and pass arguments to functions.

1. call() Method:

The call() method is used to invoke a function with a specified this value and a list of arguments. It allows you to call a function as if it were a method of a specific object, even if the function is not originally a method of that object.

Syntax: function.call(thisArg, arg1, arg2, ...);

thisArg: The object to which the this keyword will refer when the function is executed.

arg1, arg2, ...: Arguments to be passed to the function.

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
```

```
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};
const greeting = person.fullName.call(person);
console.log(greeting); // 'John Doe'
```

In this example, call() is used to execute the fullName function with the person object as the context.

2. apply() Method:

The apply() method is similar to call() but accepts arguments as an array or an array-like object.

Syntax: function.apply(thisArg, [argsArray]);

thisArg: The object to which the this keyword will refer when the function is executed.

argsArray: An array or array-like object containing the arguments to be passed to the function.

Example:

```
function sum(a, b) {
  return a + b;
}
const args = [3, 4];
const result = sum.apply(null, args);
console.log(result); // 7
```

In this example, apply() is used to call the sum function with arguments from the args array.

3. bind() Method:

The bind() method creates a new function that, when called, has its this value set to a specific object, and it can also pre-fill arguments. Unlike call() and apply(), bind() does not immediately execute the function; instead, it returns a new function with the specified context and argument values.

Example:

```
const greet = function() {
  console.log(`Hello, ${this.name}`);
};
const person = {
  name: 'Alice',
};
const greetAlice = greet.bind(person);
greetAlice(); // 'Hello, Alice'
```

Key Differences:

call() and apply() immediately invoke the function, while bind() returns a new function without executing it.

Set Object:

A Set is a collection of unique values, which means that each value can appear only once in a Set. Sets are particularly useful when you need to keep track of a list of values without any duplicates.

Map Object:

A Map is a collection of key-value pairs, where each key can be associated with a value. Maps are versatile and useful when you need to store and retrieve data based on a specific key.

Key Differences:

Sets are collections of unique values, while Maps are collections of key-value pairs.

Sets are primarily used when you need to maintain a list of unique values, whereas Maps are used when you want to associate data with specific keys.

Sets only store values, whereas Maps store both keys and values, allowing you to look up values based on their corresponding keys.

Both Sets and Maps maintain the order of insertion, meaning the order in which values or key-value pairs are added is preserved.

Sets have methods like add(), has(), and delete(), while Maps have methods like set(), get(), has(), and delete() for managing their contents.