JavaScript

JavaScript is an interpreted programming language that is primarily used for developing web applications.

It is one of the most popular and widely used languages for web development, and it is mainly run in the user's web browser.

Although JavaScript was originally created to add interactivity and dynamism to web pages, it is now used for a wide variety of purposes, including the development of mobile applications, the creation of games, and the performance of tasks on the server side.

JavaScript can be used alongside other technologies such as HTML and CSS to create rich and appealing user experiences on the web.

1. Spread Operator

Useful to combine arrays or objects

```
// Combine arrays
const arrA = [1,2,3];
const arrB = [4,5,6];
```

```
//         spread operator '...'
const arrC = [...arrA, ...arrB] // [1,2,3,4,5,6]

// Combine objects
const objA = { name: 'Elliot' }
const objB = { lastname: 'Alderson', age: 28 }

//         spread operator '...'
const objC = { ...objA, ...objB }
// { name: 'Elliot', lastname: 'Alderson', age: 28 }
```

2. Destructuring

Useful to extract data from objects and arrays

```
// Destructuring arrays
const arrA = ['pen', 'apple', 'apple-pen'];
const [ val1, val2, val3 ] = arrA;

console.log(val1); // 'pen'
console.log(val2); // 'apple'
console.log(val3); // 'apple-pen'

// Destructuring objects
const objA = { math: 97, music: 78, english: 79 };
const { math, english } = objA;

console.log(math); // 97
console.log(english); // 79
```

3. Hoisting

Hoisting is a behavior in JavaScript where variable and function declarations are "*hoisted*" to the beginning of the code block or the beginning of the JavaScript file. This means that even if you have written a variable or function declaration anywhere in your code, the JavaScript interpreter will treat it as if it was at the beginning of the code block or file.

Hoisting only applies to variable and function declarations, not variable assignments or function expressions assignments.

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

```
console.log(x) // undefined
x = 5
```

```
console.log(x) // 5
var x = 'hello'
```

For the previous example, the code will be pass to this:

```
var x = undefined;
console.log(x) // undefined
x = 5;
console.log(x);
x = 'hello'
```

4. const, let and var

- `var` declarations are hoisted and then auto-initialized to *undefined*
- `const` and `let` are hoisted too, but not initialized, so you can't access them before initialization
- `const`: declares a constant variable, which means it cannot be reassigned or redefined once it has been assigned a value. Constants must have a value assigned at the time of their declaration.
- `let`: declares a variable that can be reassigned and redefined later in the code. Unlike variables declared with `var`, variables declared with `let` are subject to the TDZ (Temporal Dead Zone). This means that they cannot be accessed or assigned until they have been assigned a value.
- `var`: declares a variable that can be reassigned and redefined later in the code. Unlike variables declared with `let` or `const`, variables declared with `var` are not subject to the TDZ and can be accessed and assigned at any time during the execution of the code. In addition, variables declared with `var` have different scope behavior than `let` and `const`. Variables declared with `var` have either global or function scope, while variables declared with `let` and `const` have block scope.

> TDZ (Temporal Dead Zone) is a JavaScript concept that refers to a stage during the execution of a program in which a variable declared with the `let` or `const` keyword is in a state of "undefined" temporarily.

5. Scope

5.1. global scope and local scope

In JavaScript, the term "scope" refers to the visibility and accessibility of variables within a program.

There are two types of scope in JavaScript: **global scope** and **local scope**.

It has the security benefit that variables can only be accessed from only a certain area of the code

- **scope global**:
  - Refers to variables that are available to be accessed from anywhere in the program.
  - These variables are defined outside of any function or block of code and are often referred to as global variables.
- **scope local**:

- Refers to variables that are only available within a specific block of code or function.
- These variables are defined within a function or block of code and are often referred to as local variables.

```
// This variable lives in the global scope
let name = 'Angela Moss';

function coolHello() {
  // This variable lives in the local scope
  // of the 'coolHello' function
  let greeting = 'Hello, my name is'

  // The 'name' variable can be used because
  // it has been declared in the global scope
  return greeting + name;
}
```

5.2. block scope y nested scope

Block scope and **nested scope** are not considered separate types of scope in JavaScript. Instead, they are ways in which the existing global and local scope can be used and applied within a program.

- **block scope**:
  - Refers to the visibility and accessibility of variables within a block of code, which is a section of code surrounded by curly braces "{ }"..
  - `let` and `const` allow you to define variables that are only available within the block of code in which they are defined.

```
console.log(x) // undefined

if (true) {
  var x = 10
  console.log(x) // 10
}

console.log(x) // 10
```

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

if (true) {
  let x = 10
  console.log(x) // 10
}

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

- **nested scope**:
  - Refers to the concept of having multiple levels of block scope within a program..
  - For example, you can have a block of code within a block of code, creating a nested block of code.

```
if (true) {
  // This variable is only available within this block of code
  let message = 'Hello world!';
  if (true) {
    // This variable is only available within this nested block of code
    let nestedMessage = 'Hello world 2';
    console.log(message); // 'Hello world!'
  }

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

6. Data Types

These are the types of data that a variable can have. There are 9 data types in JavaScript.

Seven **primitive data types**:

- `undefined`: value assigned to variables that have only been declared and do not have a value.
- `boolean`: logical data type that can have two values: true or false.
- `number`: numeric data type.
- `string`: sequence of characters used to represent text.
- `bigint`: numeric data type that can represent very large integers.
- `symbol`: allows us to obtain values that cannot be created again. They are unique and immutable identifiers.
- `Null`: represents a reference that points, usually intentionally, to a non-existent value.
- `Object`: more complex structure.
  - All other types are called "primitives" because their values can contain a single thing (whether it's a string, a number, or whatever). On the other hand, objects are used to store collections of data and more complex entities.
- `Function`: code fragment that can be called by other code or by itself. These values can be used as identifiers (keys) of the properties of objects.

> It is important to note that the `null` value is not the same as `undefined`. `null` is used to represent an intentionally non-existent value, while `undefined` is used to represent a variable that has not been initialized or an object property that does not exist.

7. Closures

A *closure* is a function that is defined inside another function and has access to the variables and parameters of the parent function, even after the parent function has finished executing.

```
function makeCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  }
}

let counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

```
function coolCounter(num, addition) {
  function add() {
    return num += addition;
  }
  return add;
}

let counter = coolCounter(4, 2);
console.log(counter()) // 6
console.log(counter()) // 8
console.log(counter()) // 10
```

8. Array methods

JavaScript arrays have several built-in methods that allow you to manipulate the elements of the array. Here are some of the most commonly used array methods:

- `.push()`: adds one or more elements to the end of an array and returns the new length of the array
- `.pop()`: removes the last element from an array and returns that element
- `.shift()`: removes the first element from an array and returns that element
- `.unshift()`: adds one or more elements to the beginning of an array and returns the new length of the array
- `.slice()`: returns a new array that includes a specified portion of an array
- `.splice()`: adds or removes elements from an array and returns the removed elements
- `.sort()`: sorts the elements of an array in place and returns the sorted array
- `.reverse()`: reverses the order of the elements of an array in place and returns the reversed array
- `.concat()`: returns a new array that combines the elements of two or more arrays
- `.join()`: returns a string that is the result of concatenating all the elements of an array, separated by a specified separator string

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

// Add a new element to the end of the array
```

```
fruits.push('mango'); // ['apple', 'banana', 'orange', 'mango']

// Remove the last element from the array
let removed = fruits.pop(); // removed = 'mango', fruits = ['apple', 'banana',
'orange']

// Add a new element to the beginning of the array
fruits.unshift('strawberry'); // ['strawberry', 'apple', 'banana', 'orange']

// Remove the first element from the array
removed = fruits.shift(); // removed = 'strawberry', fruits = ['apple', 'banana',
'orange']

// Sort the elements of the array in place
fruits.sort(); // ['apple', 'banana', 'orange']

// Reverse the order of the elements of the array in place
fruits.reverse(); // ['orange', 'banana', 'apple']
```

9. Arrays: higher-order functions

JavaScript arrays have several built-in methods that allow you to perform operations on the elements of the array and return a new array. These methods are called array functions or higher-order functions. Here are some examples of common array functions:

- `.map()`: creates a new array by applying a function to each element of the original array
- `.filter()`: creates a new array with elements that pass a test specified by a function
- `.reduce()`: reduces the elements of an array to a single value by applying a function to each element in turn
- `.forEach()`: executes a provided function once for each element in the array
- `.every()`: checks if every element in the array passes a test specified by a function and returns a boolean value
- `.find()`: returns the first element in the array that passes a test specified by a function, or undefined if no element passes the test
- `.some()`: checks if at least one element in the array passes a test specified by a function and returns a boolean value

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

// Create a new array with the squares of the numbers
let squares = numbers.map(n => n * n); // squares = [1, 4, 9, 16, 25]

// Create a new array with only the even numbers
let evens = numbers.filter(n => n % 2 == 0); // evens = [2, 4]

// Reduce the array to the sum of its elements
let sum = numbers.reduce((acc, n) => acc + n, 0); // sum = 15

// Execute a function for each element in the array
```

```
numbers.forEach(n => console.log(n)); // logs 1, 2, 3, 4, 5 to the console

// Check if all elements in the array are greater than 0
let allPositive = numbers.every(n => n > 0); // allPositive = true

// Find the first element in the array that is divisible by 2
let firstEven = numbers.find(n => n % 2 == 0); // firstEven = 2

// Check if any element in the array is divisible by 3
let anyMultipleOf3 = numbers.some(n => n % 3 == 0); // anyMultipleOf3 = true
```

10. Callbacks

A callback in JavaScript is a function that is passed to another function as an argument, and is executed after the main function has finished.

```
function myFunction(a, function myFunction(callback) {
  // code for the main function
  console.log("The function has finished");

  // execute the callback
  callback();
}

// call the function and pass an anonymous function as the callback
myFunction(function() {
  console.log("This message will be shown after the main function has finished");
});
```

Callbacks are useful because they allow functions to be executed at the right time, rather than having to wait for one function to finish before executing another.

11. Asynchrony

Asynchrony refers to not doing all tasks at the same time, i.e., doing several tasks in parallel or even if they go sequentially you should not wait to be able to continue doing something else.

Synchronous code (Blocking): Python

```
import requests

res = requests.get('https://api.com/endpoint')
print(res.text)
print('Printed after request')
```

Asynchronous code (Non Blocking): JavaScript

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

request('https://api.com/endpoint', (err, res, body) => {
  console.log(body);
});

console.log('Printed after request');
```

12. Promises

- Promises are a way to handle the result of asynchronous operations.
- A promise represents the result of an operation that has not yet completed.
- They allow you to do other things while you wait for a task to complete, and notify you when the task is ready so that you can handle the result appropriately

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    // make a call to an external API
    fetch('https://api.com/endpoint')
      .then(response => response.json())
      .then(data => resolve(data))
      .catch(error => reject(error));
  });
}

// use the promise
fetchData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

13. async-await

`async` and `await` are keywords used to work with asynchronous functions. The `async` keyword is used to define an asynchronous function. It is placed before the function declaration, like in the following example:

```
async function myAsyncFunction() {
  // function code
}

const anotherAsyncFunction = async () => {
  // function code
}
```

The `await` keyword is used inside an asynchronous function to wait for a promise to be fulfilled. For example:

```
async function myAsyncFunction() {
  const result = await someAsyncOperation();
  // continue executing the function using the result of the asynchronous
operation
}
```

When using `await`, the function execution is paused until the promise is resolved. Once the promise is resolved, the promise value is assigned to the `result` variable and the function execution continues after the line where `await` is used.

> It's important to note that `await` can only be used inside an asynchronous function. If you try to use it outside of an async function, you'll get an error.

14. Event Loop

The event loop is a mechanism that controls the execution of tasks and events in JavaScript. It's like a coordinator that makes sure everything runs smoothly in the language.

When you need to perform a task in JavaScript, it might take some time to complete. For example, if you want to load data from a server, it can take a while. Instead of having to wait for the task to complete, the event loop takes care of running the task in the background while you do other things. That way, you can keep working while the task is being completed.

15. this

In JavaScript, the `this` keyword refers to the object that owns the method in which it is being used. It can be useful for accessing the properties and methods of that object in a more concise way.

```
const name = 'Peter'

const obj = {
  name: 'Tony',
  getName: function() {
    return this.name // refers to the 'name' property of the object
  },
}

console.log(obj.getName()) // 'Tony'
```

16. bind, call and apply

16.1. bind

This method creates a new function with the same code as the original function, but with the value of `this` set explicitly. The new function can be called later and the value of `this` will remain fixed, regardless of the context in which it is called. For example:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  getName() {
    return this.name;
  }
}

function greeting() {
  console.log('Hello, my name is ' + this.getName())
}

// An instance of the Person class is created with the name "John"
const person = new Person('John');

// The "bind" method of the "greeting" function is used to bind the "person"
object as the value of "this" in the function
// The "bind" method returns a new function with the "person" object bound as the
value of "this"
const sayHi = greeting.bind(person);

// The "sayHi" function is called, which is the "greeting" function with the
"person" object bound as the value of "this"
sayHi();
```

16.2. call

This method calls a function and sets the value of `this` explicitly. It also allows you to pass parameters to the function dynamically. For example:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  getName() {
    return this.name;
  }
}

function greeting(text) {
  console.log(text + this.getName())
}

const person = new Person('John');

greeting.call(person, 'Hello, my name is ') // 'Hello, my name is John'
```

16.3. apply

This method works similarly to `call`, but instead of passing the parameters to the function as separate arguments, you pass an array with the parameters. For example:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  getName() {
    return this.name;
  }
}

function greeting(text) {
  console.log(text + this.getName())
}

greetingA.apply(person, ['Hi, my name is ']) // 'Hello, my name is John'
```