MEA - Makers a new approach to Electronic Applications

JavaScript

Riccardo Berta

2025.11.12

Contents

1 Javascript

JavaScript was created in 1995 at Netscape Communications, at that time the most popular Internet browser. Initially, it was designed **to add interactivity to web pages**, and it quickly became an essential tool for web development. Over time, JavaScript's capabilities expanded beyond simple client-side scripting, with the introduction of frameworks and libraries, and the development of Node.js, which allowed it to run on servers as well. This made JavaScript **a full-stack language**, growing its popularity even further. Adopted by all major browsers, JavaScript became standardized under the **ECMA specification**, ensuring uniformity and stability across different environments. Interestingly, despite its name, JavaScript has **nothing to do with Java**, the name was largely a marketing decision at the time to capitalize on Java growing popularity.

JavaScript is particularly well-suited for the Web of Things domain due to its versatility, real-time capabilities, and large ecosystem of libraries and tools. For instance, JavaScript can handle both front-end and back-end tasks, making it an ideal choice for handling data from various devices in a unified application. It can interact with hardware through APIs, process data efficiently, and integrate with cloud services, offering seamless communication between connected devices. Additionally, its asynchronous nature, supported by promises and async/await, is key in managing multiple IoT devices without blocking operations, making it a powerful tool for the Web of Things.

1.1 Basic features

JavaScript supports all the basic functionalities that you would expect from a programming language like C, so much of what you know is still valid. For example, comments in JavaScript are written using the same syntax as in C:

```
// This is a comment
```

Blocks of code are enclosed within curly braces, just like in C. Conditional execution works in the same way, with the basic constructs like "if", "if...else", and "switch...case":

```
if (x > 10) {
    console.log("x is greater than 10");
} else {
    console.log("x is less than or equal to 10");
}
```

To run JavaScript code directly in our browser, such as Google Chrome, we can use the **built-in Developer Tools** on the **Console** tab. This is where we can enter and run JavaScript code using the internal brwoser JavaScript interpreter.

Loops are also supported, including ”while”, ”do...while”, and ”for”, as well as ”for-in” for iterating over objects or arrays:

```
let number = 1;

while (number ⩽ 12) {
    console.log(number);
    number++;
}
```

A fragment of code that produces a value is called an **expression** and a statement, on the other hand, is a **composition of expressions** and ends with a semicolon:

```
// This is an expression that produces a value
let result = 5 * 3;
```

The semicolon used to mark the end of a statement, it is technically optional in many cases, due to a feature called **Automatic Semicolon Insertion**). It allows JavaScript to automatically insert semicolons where it assumes they should go, so you don’t always need to explicitly write them. However, there are cases where ASI might not behave as expected, leading to bugs. To avoid unexpected behavior and make the code clearer to both the browser and other developers, it’s generally recommended to always use semicolons to explicitly mark the end of statements.

Variables can be declared using **let**, **const**, or **var** (though let and const are preferred in modern JavaScript). Functions are also defined similarly to how you would do so in C, allowing us to structure and reuse our code effectively:

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

So, since you are already familiar with the basics of C, you’ll find JavaScript to be quite similar in its structure and functionality.

1.1.1 Variables

Of course there are important differences between C and JavaScript, one of the most notable being the way variables are declared. In C, you would declare a variable by specifying its type, such as int, float, or char. Instead, JavaScript is a **dynamically typed language**, meaning that we don't need to declare the type of a variable explicitly. Instead, the **type is determined by the value assigned to it at runtime**. For example, when we write:

```
let name = value;
```

JavaScript automatically infers the type of name based on the type of value. So we don't have to declare whether a variable should hold a string, number, or any other type. A key consequence is that **variables can hold values of different types at different times**. For instance, we might assign a string to a variable and later change it to a number:

```
// x is a string
let x = 'pippo';

// now x is now a number
x = 5;
```

This flexibility allows for greater ease of use, but it can also lead to **automatic type conversion**, where JavaScript implicitly changes the type of a value in certain operations. This behavior can be **confusing** and lead to unexpected results:

```
console.log('5' - 1);  // 4 (string '5' is converted to number)
console.log('5' + 1);  // '51' (number 1 is converted to string and concatenated)
console.log('5' * 2);  // 10 (string '5' is converted to number and multiplied)
```

For example, in the expression '5' - 1, JavaScript will convert the string '5' to a number and subtract 1, resulting in 4. However, in the expression '5' + 1, JavaScript treats the + operator as a string concatenation operator when one of the operands is a string. As a result, the number 1 is automatically converted to a string, and the result is '51' instead of performing numeric addition. This automatic conversion can sometimes be hard to track, especially when the operations involve different data types.

While dynamic typing provides a lot of flexibility, it can sometimes make the code harder to understand and debug, especially when type conversions happen automatically and unexpectedly. Developers need to be cautious and often explicit about the types they expect in their code to avoid tricky bugs.

1.1.2 Functions

Another important key feature of JavaScript is its support for **first-class functions**. A function, like in C, is a block of code designed to perform a specific task or computation. It takes input, processes it, and produces an output. Functions help in organizing code, making it more reusable and modular. Instead of writing the same code repeatedly, we can define it once inside a function and call it whenever we need it. In JavaScript, a function is typically defined using the "function" keyword and can have parameters to represent the values we pass into the function and a body, which contains the code that defines what the function does. For example, a simple function that adds two numbers could look like this:

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

In this example, a and b are parameters, and a + b is the computation the function performs. When the function is called, we provide arguments that correspond to the parameters, like so:

```
let result = add(3, 4);  // result is now 7
```

Functions can return values, or they can perform actions without returning anything. If a function doesn't explicitly return a value, it implicitly returns **undefined**.

The specificity of JavaScript is that funntions are treated like any other value, such as numbers or strings. A function can be assigned to a variable, passed around, or reassigned just like any other value. For example, we can define a function and assign it to a variable like this:

```
let name = function(parameters) {
    // body
};
```

Since functions are just variables, they can be used in any expression and can even be assigned to new values. We could, for example, reassign square to another function or value entirely:

```
name = function(parameters) {
    // another body;
};
```

Another interesting feature of functions in JavaScript is **nested scope**. A nested function is simply a function defined inside another function. The inner function is **accessible only** within the outer function and is useful for modularizing code, improving readability, and encapsulating logic that does not need to be exposed globally:

```
function outerFunction(a, b) {
    console.log("Outer function executing...");

    function innerFunction(x, y) {
        return x + y;
    }

    // Call inner function
    let result = innerFunction(a, b);

    console.log("Result:", result);
}

// call the outer function that will call the inner function
outerFunction(5, 3);
```

JavaScript also supports **arrow functions**, which provide a more concise syntax for defining functions. Arrow functions are particularly useful for short, one-line functions. They are defined using the **=>** syntax and do not require the function keyword:

```
// No Parameters → Use empty parentheses:
() ⇒ { console.log("Hello!"); }

// One Parameter → Parentheses are optional:
value ⇒ console.log(value);
(value) ⇒ console.log(value);

// Two or More Parameters → Parentheses are required:
(a, b) ⇒ a + b;
```

They are more concise, reducing **boilerplate code** and making function expressions easier to read and write. When the function consists of a single expression, the curly braces and return keyword can be omitted, leading to a more streamlined syntax:

```
// Implicit return (when there's only one expression)
let anon = (a, b) ⇒ a + b;
console.log(anon(2, 3)); // 5

// Explicit return using curly braces
let anon = (a, b) ⇒ { return a + b };
```

Summarizing, functions can be assigned to variables, passed as arguments to other functions, and returned from functions. This flexibility allows for powerful programming paradigms like **functional programming** and **callbacks**.

1.1.3 Equality

We need to pay attention to equality comparisons, as they can behave differently from what we might expect coming from C. In JavaScript, there are two types of equality comparisons: **loose equality** (==) and **strict equality** (===).

The **strict equality operator** (===) checks both the type and the value of the operands to ensure they are exactly the same. For example:

```
5 === 5 // true, both are numbers with the same value
'hello world' === 'hello world' // true, both are strings with the same value
77 === '77' // false, one is a number, the other is a string
false === 0 // false, different types (boolean vs. number)
```

The **loose equality operator** (==), on the other hand, performs type **coercion**. It tries to convert the operands to a common type before comparing their values. For example:

```
77 == '77' // true, string '77' is coerced to the number 77
false == 0 // true, false is coerced to 0
null == undefined // true, they are considered equal in loose equality
```

While == may seem more flexible, the triple equals (===) operator is generally preferred because it ensures a more reliable equality check by comparing both the type and the value of the operands. This helps avoid unexpected behavior due to type coercion:

```
// Correct way to check for equality
'123' === 123 // false, different types (string vs. number)
```

```
'123' == 123 // true, type coercion happens, '123' is converted to 123

// Best practice: Always use ===
if (value === 5) {
    console.log("The value is exactly 5");
}
```

Using strict equality (===) ensures that you're performing a true equality test, avoiding the pitfalls of implicit type conversions.

1.1.4 Const, let and var

In JavaScript, the way variables are declared has evolved, especially with the introduction of **let** and **const** in ES6. Before ES6, the **var** keyword was the only way to declare a variable, but it comes with some quirks that can lead to unexpected behavior. For example, variables declared with var are **function-scoped, not block-scoped**. This means they are accessible throughout the function, regardless of where they are declared within a block, like inside a loop or an if statement:

```
function testVar() {
    if (true) {
        // `x` is available throughout the entire function, not just the block
        var x = 10;
    }

    // Outputs 10, even though `x` was declared inside the `if`
    console.log(x);
}
```

This can lead to problems if we accidentally override a variable within a block, making the **behavior harder to predict**. Additionally, var is **hoisted**, meaning we can use a variable before it is declared! However, only the declaration is hoisted, not the assignment.

```
// Outputs undefined, due to hoisting
console.log(y);
var y = 20;
```

With ES6, let and const were introduced to address some of these issues. let is **block-scoped**, which means it is limited to the block in which it is declared, making it safer for use inside loops or conditionals:

```
function testLet() {
    if (true) {
        // `z` is only available within this block
        let z = 30;
        console.log(z);
    }
    // Error: `z` is not defined outside the block
    console.log(z);
}
```

const is also block-scoped but, unlike let, it makes the variable **read-only**. Once a variable is assigned with const, it cannot be reassigned, ensuring that its value remains constant.

```
const name = "John";

// Error: Assignment to constant variable
name = "Doe";
```

In most cases, it's recommended to use const for variables that will not be reassigned, as it helps prevent accidental changes and improves code clarity. Use let only when the value of a variable is expected to change, such as in loops or when the variable needs to be dynamically updated.

```
// Use const for constants
const pi = 3.14159;

// Use let when the value will change
let radius = 10;

// This is fine since we're using `let`
radius = 15;
```

As for var, it is best avoided in modern JavaScript, as let and const provide better, more predictable scoping behavior.

1.1.5 Arrays

Arrays in JavaScript are similar to arrays in C, but they are more flexible and dynamic. Arrays allow us to **store multiple values** within a single variable and are often used to represent **lists** or **collections** of data. For example, you can store a list of numbers like this:

```
let listOfNumbers = [2, 3, 5, 7, 11, 4];
console.log(listOfNumbers[2]); // Outputs 5
```

In the example, the array contains six elements. Arrays in JavaScript are indexed, meaning each element can be accessed by its index, starting from 0. So, listOfNumbers[2] accesses the third element.

Arrays in JavaScript have various **properties** that provide useful information or functionality. One important property is **length**, which returns the number of elements in the array:

```
console.log(listOfNumbers.length); // Outputs 6
```

When accessing properties of an array, we can use either **dot notation** or **bracket notation**. The bracket notation allows use to use expressions or **dynamic property names**. For instance, we can access the length property like this:

```
console.log(listOfNumbers['length']);
```

Arrays also come with several** built-in methods for manipulating the elements**. For example, we can** add or remove elements at the end** of an array using push() and pop():

```
listOfNumbers.push(13);  // Adds 13 to the end of the array
console.log(listOfNumbers); // Outputs [2, 3, 5, 7, 11, 4, 13]

listOfNumbers.pop();  // Removes the last element (13)
console.log(listOfNumbers); // Outputs [2, 3, 5, 7, 11, 4]
```

Similarly, we can **add or remove elements at the start** of an array using unshift() and shift():

```
listOfNumbers.unshift(1);  // Adds 1 to the beginning of the array
console.log(listOfNumbers); // Outputs [1, 2, 3, 5, 7, 11, 4]

listOfNumbers.shift();  // Removes the first element (1)
console.log(listOfNumbers); // Outputs [2, 3, 5, 7, 11, 4]
```

We can also **search for elements** in an array using indexOf() and lastIndexOf(). These methods return the index of the first and last occurrence of a specified element, respectively:

```
console.log(listOfNumbers.indexOf(5)); // Outputs 2, the index of 5
console.log(listOfNumbers.lastIndexOf(7)); // Outputs 3, the last index of 7
```

Another useful method is slice(), which allows you to **get a portion of an array** between two specified indexes, without modifying the original array:

```
let slicedArray = listOfNumbers.slice(1, 4); // Extracts elements from index 1 to 3
 ↪  (4 is exclusive)
console.log(slicedArray); // Outputs [3, 5, 7]
```

Arrays in JavaScript are **dynamic** and **versatile**, making them an essential tool for storing and managing collections of data. The various properties and methods allow us to perform a wide range of operations, from accessing and modifying elements to searching and slicing arrays.

1.1.6 Strings

Strings in JavaScript are similar to strings in C, but they come with additional features and methods that make them more powerful. Strings are used to represent text data and are enclosed in either single quotes (') or double quotes (""). For example:

```
let carName1 = "Volvo XC60";
let carName2 = 'Volvo XC60';
```

Both variables hold the same value, but we can use whichever type of quotes suits our need or preference. Additionally, we can include quotes inside a string, as long as the quotes inside do not match the ones that are enclosing the string. For example:

```
let answer1 = "It's alright";  // Single quote inside a double-quoted string
let answer2 = "He is called 'Johnny'";  // Single quotes inside double quotes
let answer3 = 'He is called "Johnny"';  // Double quotes inside single quotes
```

Strings come with several useful **properties** and methods. For example, the **length** property returns the number of characters in a string:

```
let carName = "Volvo XC60";
console.log(carName.length);  // Outputs 12
```

We can also transform a string to **uppercase** or **lowercase**:

```
let greeting = "Hello World";
console.log(greeting.toUpperCase());  // Outputs "HELLO WORLD"
console.log(greeting.toLowerCase());  // Outputs "hello world"
```

If you need to **extract a part of a string**, we can use the slice() method, or we can **find the position of a specific character** with indexOf() or access a character directly with charAt():

```
let phrase = "Hello World";
console.log(phrase.slice(0, 5));  // Outputs "Hello"
console.log(phrase.indexOf('o'));  // Outputs 4 (index of the first 'o')
console.log(phrase.charAt(6));  // Outputs 'W' (character at index 6)
```

The trim() method is helpful for **removing any whitespace** from both ends of a string:

```
let message = "   Hello World   ";
console.log(message.trim());  // Outputs "Hello World" (without spaces at the ends)
```

When comparing strings, JavaScript **compares them based on their value**. For example:

```
let answer1 = "It's alright";
let answer2 = "It's alright";
console.log(answer1 === answer2);  // Outputs true
```

This comparison checks both the content and the type of the strings, so it returns true when the strings are identical.

1.1.7 Parameter passing

In JavaScript, primitive values such as numbers, strings, and booleans are **passed by value** when used as function arguments. This means that when a primitive is passed to a function, the function **receives a copy of the value rather than the original**. Any modifications made inside the function do not affect the original variable outside the function:

```
function no_modify_primitives(x) {
    x = 7;
}
```

```
let x = 3;
no_modify_primitives(x);

// Outputs 3, because x was passed by value
console.log(x);
```

For arrays (and objects), JavaScript follows a **call-by-sharing** strategy, which is often confused with **pass-by-reference**. When an array is passed to a function, the function receives a reference to the same memory location rather than a copy of the value. This allows the function to modify the contents of the array or object:

```
function modify_array(array) {
    array[0] = 7;
}

let a = [3, 0, 2];

modify_array(a);

// Outputs [7, 0, 2], since the modification affects the original array
console.log(a);
```

However, if a function tries to assign a new array to the parameter, the **reference is broken**, and the function starts working with a new local version. This means that the original array or object remains unchanged outside the function.

```
function no_modify_array(array) {
    // This creates a new array, breaking the reference
    array = [3, 0, 2];
    array[0] = 9;
}

no_modify_array(a);

// Outputs [7, 0, 2], because reassignment inside the function
// does not affect the original array
console.log(a);
```

This behavior is crucial to understand when working with functions that modify arrays or objects, as direct modifications will persist, but reassignment will not affect the original variable.

1.1.8 Optional arguments

JavaScript is **very flexible** when it comes to function arguments. If you pass more arguments than a function expects, the extra ones are simply **ignored**. If you pass fewer than required, the missing arguments are assigned the value **undefined**. This behavior allows functions to handle optional parameters, but it can also lead to unintended bugs if a function is accidentally used incorrectly. A common approach to dealing with optional arguments is to assign default values when an argument is undefined. This allows the function to work even when some parameters are omitted.

```
function power(base, exponent) {
    if (exponent == undefined) exponent = 2;
    let result = 1;
    for (let i = 0; i < exponent; i++) result *= base;
    return result;
}

console.log(power(3, 3)); // Outputs 27 (3^3)
console.log(power(3));    // Outputs 9  (3^2), using the default exponent
```

With the introduction of ES6, JavaScript provides a more concise way to handle optional arguments using default parameters:

```
function power(base, exponent = 2) {
    let result = 1;
    for (let i = 0; i < exponent; i++) result *= base;
    return result;
}

console.log(power(3, 3)); // Outputs 27
console.log(power(3));    // Outputs 9
```

Using default parameters directly in the function definition makes the code cleaner and easier to read while avoiding explicit checks for undefined. This feature is particularly useful when designing functions that can handle various use cases with minimal redundancy.

1.1.9 Recursion

Recursion is a programming technique where **a function calls itself** to solve a problem by breaking it down into smaller subproblems. In JavaScript, recursion is often used when a problem

can be naturally divided into simpler instances of itself, such as mathematical calculations, tree traversal, or searching algorithms. A classic example of recursion is computing the power of a number. Instead of using a loop, we can define a function that calculates the exponentiation of a base number by multiplying it with a recursive call to itself with a reduced exponent:

```
function power(base, exponent) {
    if (exponent == undefined) exponent = 2;
    if (exponent == 0) return 1;
    else return base * power(base, exponent - 1);
}

console.log(power(3, 3)); // Outputs 27 (3^3)
console.log(power(3));    // Outputs 9  (3^2, using the default exponent)
console.log(power(2, 5)); // Outputs 32 (2^5)
console.log(power(5, 0)); // Outputs 1  (base case)
```

In this function, the base case is when exponent reaches 0, in which case the function returns 1, as any number raised to the power of zero equals one. If the exponent is greater than zero, the function multiplies base by the result of calling power(base, exponent - 1), gradually decreasing exponent until it reaches the base case.

Each recursive call reduces the problem's complexity until it can be directly solved. This approach eliminates the need for explicit loops, making the code more readable and elegant. However, recursion should be used carefully, as excessive recursion depth can lead to stack overflow errors if the function calls itself too many times without reaching a termination condition.

1.1.10 Error Handling

Error handling is a crucial aspect of programming, ensuring that programs can **respond to unexpected situations** or faults in a controlled manner. In JavaScript, error handling allows developers to catch and manage errors during runtime, preventing the application from crashing and providing users with meaningful feedback. JavaScript provides a robust system for error handling, relying heavily on the **try**, **catch**, and **finally** keywords to structure error management. The try block contains the code that might throw an error. If an error occurs within this block, JavaScript immediately jumps to the catch block, where the error can be handled. This allows the program to continue running without interruption. The catch block can access the error object, which contains details about the error, such as the message, stack trace, and error type. This makes it possible to handle different types of errors appropriately and provide relevant feedback. For instance, in a situation where we're trying to parse a string into JSON, a potential

error could occur if the string is not valid JSON. Using a try...catch block, we can gracefully handle this:

```
try {
    let data = JSON.parse('{"name": "John", age: 30}'); // Syntax error
}
catch (error) {
    console.log("An error occurred:", error.message);
}
```

Additionally, JavaScript provides a finally block, which is optional. Code inside this block runs regardless of whether an error occurred or not. It is often used for cleanup tasks, such as closing files or releasing resources, ensuring that these actions are always performed no matter what.

```
try {
    let result = performRiskyOperation();
}
catch (error) {
    console.error("Error:", error);
}
finally {
    cleanup();
}
```

When an error occurs, JavaScript generates an object containing the details about it. This object has several properties, such as message, name, and stack, which can be used to identify the error and provide useful information to the developer. For example, the message property contains a human-readable description of the error, while the stack property contains a stack trace showing the sequence of function calls that led to the error

```
try {
    lalala; // This will cause a ReferenceError
}
catch (err) {
    alert(err.name); // ReferenceError
    alert(err.message); // lalala is not defined
    alert(err.stack); // ReferenceError: lalala is not defined at ( ... call stack)

    // Can also show an error as a whole
    // The error is converted to string as "name: message"
```

```
    alert(err); // ReferenceError: lalala is not defined
  }
```

1.1.11 Modules

Modules are a way **to organize and structure code** into reusable, isolated units that can be easily imported and exported between different parts of an application. The concept of modules helps manage complexity in large codebases by breaking down functionality into smaller, self-contained pieces. Each module can have its own set of variables, functions, and objects, which are scoped to that module and not accessible globally unless explicitly exported. For example, to export a function from a module, you can use the export keyword:

```
// module.js
export function greet(name) {
    return `Hello, ${name}!`;
}
```

Then, in another file, we can import and use the function like this:

```
// app.js
import { greet } from './module.js';

console.log(greet('Alice'));  // Outputs: Hello, Alice!
```

In the example, the greet() function is exported from module.js and imported into app.js. The function is then called within the application, demonstrating how different parts of the application can share code through modules.

Modules also provide a way to control what is exposed to other parts of the program. By default, only the exported members of a module are accessible outside of it. This encapsulation helps maintain cleaner and more maintainable code by preventing unwanted side effects from global variables or functions. Additionally, a module can export multiple values at once, or even the default value for a module can be exported, making it easier to import without needing to reference specific exports.

```
// module.js
export const name = 'JavaScript';
export function greet(name) {
```

```
  return `Hello, ${name}!`;
}
```

```
// app.js
import * as utils from './module.js';

console.log(utils.name);  // Outputs: JavaScript
console.log(utils.greet('Alice'));  // Outputs: Hello, Alice!
```

JavaScript also supports CommonJS and AMD modules, which are widely used in environments like Node.js and older web applications, respectively. These module systems offer their own ways to import and export code, with CommonJS using require() and module.exports, and AMD using define() and require().

1.2 Code examples

In this section, we will explore a range of fundamental algorithms implemented using JavaScript.

1.2.1 Factorial of a number

The factorial of a number is the product of all positive integers less than or equal to that number. For example, the factorial of 5 (written as 5!) is 5 * 4 * 3 * 2 * 1 = 120. We can calculate the factorial of a number using a recursive function in JavaScript:

```
function factorial(x) {
    if (x < 0) throw new Error("Factorial is not defined for negative numbers.");
    if (x === 0 || x === 1) return 1;
    return x * factorial(x - 1);
}

console.log(factorial(15)); // Outputs 1307674368000
console.log(factorial(0));  // Outputs 1
console.log(factorial(1));  // Outputs 1
```

1.2.2 Binary search

Binary search is an efficient algorithm for finding a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval

is empty. Here's an example of a binary search function in JavaScript:

```
function binarySearch(array, target, start = 0, end = array.length - 1) {
    // Base case: not found
    if (start > end) return undefined;

    const mid = Math.floor((start + end) / 2);

    if (array[mid] === target) return mid;
    if (array[mid] < target) return binarySearch(array, target, mid + 1, end);

    return binarySearch(array, target, start, mid - 1);
}

const list = [21, 45, 76, 93, 204, 345, 6654];

console.log(binarySearch(list, 345)); // Outputs: 5
console.log(binarySearch(list, 50));  // Outputs: undefined (not found)
console.log(binarySearch(list, 21));  // Outputs: 0
console.log(binarySearch(list, 6654)); // Outputs: 6
```

1.2.3 Bubble sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, indicating that the list is sorted. Here's an example of a bubble sort function in JavaScript:

```
function swap(array, first_index, second_index){
    let temp = array[first_index];
    array[first_index] = array[second_index];
    array[second_index] = temp;
}

function bubble_sort(array){
    let len = array.length;
    for (let i=0; i<len; i++)
        for (let j=0; j<len-i; j++)
            if (array[j] > array[j+1]) swap(array, j, j+1);
    return array;
}
```

```
let a = [3, 0, 2, 5, -1, 4, 1];
console.log(bubbleSort(a)); // [-1, 0, 1, 2, 3, 4, 5]
```

1.2.4 Quicksort

Quicksort is a popular sorting algorithm that uses a divide-and-conquer strategy to sort an array. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. Here's an example of a quicksort function in JavaScript:

```
function quickSort(array) {
    if (array.length <= 1) return array;

    // Take the last element as pivot
    const pivot = array[array.length - 1];

    const left = [];
    const right = [];

    for (let i = 0; i < array.length - 1; i++) {
        if (array[i] <= pivot) {
            left.push(array[i]);
        } else {
            right.push(array[i]);
        }
    }
    return quickSort(left).concat(pivot, quickSort(right));
}

let myArray = [3, 0, 2, 5, -1, 4, 1];
console.log("Original array:", myArray);

let sortedArray = quickSort(myArray);
console.log("Sorted array:", sortedArray);
```

1.3 Functional Programming

JavaScript supports **functional programming**, a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

Functional programming emphasizes the use of **pure functions**, which produce the same output for the same input and do not have side effects. This approach leads to more predictable and maintainable code, as functions are isolated and do not rely on external state. JavaScript provides several features that support functional programming, such as **higher-order functions** and **closures**.

1.3.1 High order functions

Higher-order functions are functions that either **take other functions as arguments**, **return functions**, or both. They allow for the abstraction of behavior over values, enabling a **more flexible and functional style** of programming. These functions are powerful because they allow us to write more generic, reusable, and concise code. By treating functions as first-class citizens (just like numbers or strings), we can pass them around, return them from other functions, or create specialized behavior dynamically. A higher-order function operates on other functions in a way that allows us to encapsulate specific actions, not just values.

```
function greaterThan(n) {
    return function(m) { return n > m; };
}

let greaterThan10 = greaterThan(10);
console.log(greaterThan10(7)); // true
console.log(greaterThan10(11)); // false

let greaterThan4 = greaterThan(4);
console.log(greaterThan4(3)); // true
console.log(greaterThan4(5)); // false
```

In the example above, greaterThan is a higher-order function. It returns a function that checks if n is greater than a given number m. The variable n is accessible inside the inner function due to lexical scoping, where the inner function "remembers" the environment in which it was created. This ability to return functions and capture variables from their surrounding scope is a key feature of functional programming.

1.3.2 Filter, map and Reduce

JavaScript provides several built-in higher-order functions that are commonly used in functional programming. Three of the most popular ones are **filter**, **map**, and **reduce**. These functions

allow us to manipulate arrays in a concise and expressive way, making it easier to work with collections of data.

The **filter** function takes an array and a test function as arguments, and it returns a new array containing all elements that pass the test:

```
function filter(array, test) {
    let passed = [];
    for (let i = 0; i < array.length; i++) {
        if (test(array[i])) passed.push(array[i]);
    }
    return passed;
}

let a = [2, 3, 5, 6];
let test = function(x) { return x >= 5; };
let filtered = filter(a, test);
console.log(filtered); // [5, 6]
```

The **map** function transforms each element in an array using a transformation function.

```
function map(array, transform) {
    let mapped = [];
    for (let i = 0; i < array.length; i++) {
        mapped.push(transform(array[i]));
    }
    return mapped;
}

let a = ['3', '5', '6'];
let transform = function(x) { return Number(x); };
let mapped = map(a, transform);
console.log(mapped); // [3, 5, 6]
```

The **reduce** function combines all elements of an array into a single value, using a combining function and an initial value.

```
function reduce(array, combine, start) {
    let current = start;
    for (let i = 0; i < array.length; i++) {
        current = combine(current, array[i]);
```

```
    }
    return current;
}

let a = [3, 5, 6];
let sum = function(x, y) { return x + y; };
let result = reduce(a, sum, 0);
console.log(result); // 14
```

The implementation of filter, map, and reduce for arrays is straightforward and may seem intuitive, but it is often naïve and inefficient. Manually iterating over an array and applying transformations using loops can lead to unnecessary complexity, reduced readability, and potential performance issues. JavaScript arrays already provide built-in .filter(), .map(), and .reduce() methods, which are optimized for performance.

```
let filtered = a.filter(test);
let mapped = a.map(value => value * 2);
let reduced = a.reduce(sum, 0);
```

In general, instead of manually iterating and modifying lists, leveraging language-based methods to ensure better performance and cleaner code.

1.3.3 Closures

One of the most powerful features of higher-order functions is the concept of closures. When a function is returned by another function, the inner function **retains access** to the variables of the outer function, even after the outer function has completed execution.

```
function wrapValue(n) {
    let localVariable = n;
    return function() { return localVariable; };
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);

console.log(wrap1()); // 1
console.log(wrap2()); // 2
```

In this example, wrapValue returns a function that retains access to localVariable even after the wrapValue function has finished executing. This enables a powerful pattern where we can **"freeze"** specific values for later use. This pattern is useful in **stateful applications**, like managing user sessions, caching results, or handling module states. As an explicative example, we can create a simple counter that mantains its state between calls:

```
function createCounter() {
    let count = 0;
    return function () {
      count++;
      console.log(`Current count: ${count}`);
    };
}

const counter = createCounter();

counter(); // Current count: 1
counter(); // Current count: 2
counter(); // Current count: 3
```

1.3.4 Currying

A powerful application of closures is **currying**, a technique that transforms a function from being called as $f(a, b, c)$ into a sequence of calls like $f(a)(b)(c)$, or any number of parameters. Let's explore an example to clarify the concept, followed by practical applications. We'll start by implementing a helper function $curry()$ which enables currying for a three-argument function:

```
function curry(f) {
    return function(a) {
      return function(b) {
        return function (c) {
            return f(a, b, c);
        }
      };
    };
  }

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

```
let curriedSum = curry(sum);
console.log(curriedSum(1)(2)(3));
```

The implementation is straightforward: it's just three wrappers. The result of curry() is a wrapper function(a). When it is called (like curriedSum(1)), the argument is saved in the freezed variable a, and a new wrapper is returned (function(b)). Then this wrapper is called with the second argument, and again, the argument is saved in the freezed variable b, and a new wrapper is returned (function(c)). Finally, when the last wrapper is called with the third argument, the original function is called with all three arguments. To understand the benefits of this technique, we need a real-life example. Suppose we have a logging function that formats and outputs some information, like:

```
function log(date, type, message) {
    console.log(date.getHours(), date.getMinutes(), type, message);
}
```

Let's curry the log function:

```
let curriedLog = curry(log);
```

After that log works normally, but also works in the curried form:

```
log(new Date(), "DEBUG", "Message");
curriedLog(new Date())("DEBUG")("Message");
```

However, by using the curried version, we can create a **specialized** logging function. Instead of repeatedly passing the same argument, we can define a version that automatically includes common parameters, making the code more concise. For example, we can create a function that prepends a timestamp to each message:

```
let logNow = curriedLog(new Date());
logNow("DEBUG")("Message");
```

The function logNow() is a version of the log() function with fixed first argument, in other words "partially applied function" or "partial" for short. We can go further and make a convenient function for current debug logs:

```
let debugNow = logNow("DEBUG");
debugNow("Message");
```

Higher-order functions are a cornerstone of Javascript programming. By allowing functions to operate on other functions, they enable more modular, reusable, and concise code. Whether we're using them for mapping, filtering, reducing, or creating closures and curries, higher-order functions offer a powerful toolset for abstracting and composing behaviors in our applications.

1.4 Object Oriented Programming

JavaScript is a versatile language that supports multiple programming paradigms, including **object-oriented programming** (**OOP**). OOP is a programming paradigm that organizes code into objects, which encapsulate data and behavior. Objects interact with each other through methods, and they can inherit properties and methods from other objects.

1.4.1 Objects

Objects are **arbitrary collections of properties** that can be **created directly without requiring a class**. An object is simply a set of key-value pairs, where keys are property names and values can be any type, including numbers, strings, arrays, or even other objects. The syntax is straightforward:

```
let example = { name: 'example', values: [1, 2, 3], length: 3 };
console.log(example);
```

Each property is defined by a name, followed by a colon and its corresponding value, separated by commas. Objects are flexible and can be dynamically modified by **adding or removing properties at runtime**. One important thing to remember is that objects are **reference-based**. Comparing two different objects using == or === will always return false, even if they have identical properties. This is because they are stored at different memory locations, similar to pointers in languages like C or Java.

```
let example1 = { name: 'example' };
let example2 = { name: 'example' };
console.log(example1 == example2); // false
```

Objects can also contain **methods**, which are properties that hold function values. Inside these functions, the keyword **this** refers to the object itself, allowing access to other properties. For example:

```
let example = {
    name: 'example',
    append: function(x) {
      this.name = this.name + ' ' + x;
    }
};

console.log(example);
example.append('pippo');
console.log(example);
```

JavaScript also provides built-in objects as collections of methods and properties related to specific functionalities. For example, the **Math** object contains mathematical constants and functions (like Math.sqrt(), Math.random() and Math.pow()), while the **Date** object provides methods for working with dates and times. These built-in objects are useful for common tasks and can be accessed directly without the need to create instances. In particular, it is really useful for our IoT applications to work with the **JSON** object, which provides methods for converting objects to JSON strings and vice versa.

1.4.2 Factories

Creating objects directly can be useful in several cases, however it can be cumbersome when we need to create multiple objects with similar properties and methods. In these cases, we can use **factory functions** to create objects based on a template. A factory function is a function that returns an object, allowing us to create multiple instances with shared properties and methods. For example:

```
function createUser(name, age) {
    return {
      name: name,
      age: age,
      greet() {
        console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
      }
    };
```

```
}

let user1 = createUser("Alice", 25);
let user2 = createUser("Bob", 30);

user1.greet(); // "Hello, my name is Alice and I am 25 years old."
user2.greet(); // "Hello, my name is Bob and I am 30 years old."
```

Think of a factory as an automated assembly line. Each time it is called, it produces a new object, just like a factory producing identical products with different configurations. This method is useful for encapsulation, allowing internal logic to be hidden while providing a simple interface to create objects.

1.4.3 Constructors

Another way to create objects in JavaScript is by using **constructors**. A constructor is a function that is called with the **new** keyword to create a new object:

```
function User(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
      console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
    };
}

let user1 = new User("Alice", 25);
let user2 = new User("Bob", 30);

user1.greet(); // "Hello, my name is Alice and I am 25 years old."
user2.greet(); // "Hello, my name is Bob and I am 30 years old."
```

When we use new with a constructor, JavaScript does several things behind the scenes: it creates a new empty object, binds this to that object, and then runs the constructor function, allowing us to initialize properties and methods for the object. After the constructor function executes, it automatically returns the object created. The main difference with a factory function is that all instances created with a constructor are **linked to a prototype**. The prototype is a fundamental feature of the language **inheritance model**, it is an object from which an object inherits properties and methods. When we access a property or a method on an object, JavaScript looks for it

on the object itself, and if it doesn't find it there, it looks up the prototype. This allows objects to share functionality and reduces memory usage by avoiding duplication of methods across all instances of an object. When we use a constructor function, every instance created by that constructor shares a common prototype. The prototype is typically used to define methods or properties that should be available to all instances, ensuring that these methods aren't copied for each instance, but instead are shared across them:

```
function User(name, age) {
    this.name = name;
    this.age = age;
  }

  // Adding a method to the prototype
  User.prototype.greet = function() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
};

let user1 = new User("Alice", 25);
let user2 = new User("Bob", 30);

// Both user1 and user2 share the same greet method via the prototype.
user1.greet(); // "Hello, my name is Alice and I am 25 years old."
user2.greet(); // "Hello, my name is Bob and I am 30 years old."
```

In the example, greet() is defined on the prototype of User, which means both instances (user1 and user2) share the same greet() method. This results in **memory efficiency** because the method is not recreated for each instance of User. Instead, both objects look up the method on the prototype. Factory functions, instead, don't rely on prototypes. Each time a factory function is called, it creates a completely new object that is not linked to any prototype. This means that each instance of an object created by a factory function is independent, and does not share methods via a prototype. However, factory functions provide more flexibility and control over object creation. We can encapsulate private data and logic within the factory, whereas constructors are more rigid and focus primarily on creating instances with shared prototypes. In summary, constructor functions are ideal when we want multiple instances to share common methods via the prototype, reducing memory usage and enabling inheritance. Factory functions provide more flexibility and are suitable when we need independent objects.

As an example, we can create the support fro a linked list in JavaScript using the following code (see *01.LinkedList.js* script). First we create the constructor:

```
function List() {
    this.makeNode = function() { return { data: null, next: null }; };
    this.start = null;
    this.end = null;
}
```

The we add some methods to the prototype:

```
List.prototype.add = function(data) {
    if (this.start === null) { this.start = this.makeNode(); this.end = this.start; }
    else { this.end.next = this.makeNode(); this.end = this.end.next; };
    this.end.data = data;
};
```

```
List.prototype.delete = function(data) {
    let current = this.start;
    let previous = this.start;
    while (current !== null) {
      if (data === current.data) {
        if (current === this.start) { this.start = current.next; return; }
        if (current === this.end) this.end = previous;
        previous.next = current.next; return;
      }
      previous = current;
      current = current.next;
    }
};
```

```
List.prototype.insertAsFirst = function(d) {
    let temp = List.makeNode();
    temp.next = this.start;
    this.start = temp;
    temp.data = d;
  };
```

```
List.prototype.insertAfter = function(t, d) {
    let current = this.start;
    while (current !== null) {
      if (current.data === t) {
        let temp = List.makeNode();
```

```
        temp.data = d;
        temp.next = current.next;
        if (current === this.end) this.end = temp;
        current.next = temp;
        return;
      }
      current = current.next;
    }
  };
```

```
List.prototype.item = function(i) {
    let current = this.start;
    while (current !== null) {
      i--;
      if (i === 0) return current;
      current = current.next;
    }
    return null;
  };
```

```
List.prototype.each = function(f) {
    let current = this.start;
    while (current !== null) {
      f(current);
      current = current.next;
    }
  };
```

Finally, we can create a new linked list and add some elements:

```
var list = new List();

for(var i=1;i<=10;i++){
    list.add(i);
};

list.each(function(item) { console.log(item.data); });

list.delete(4);

list.each(function(item) { console.log(item.data); });
```

1.4.4 Getter and Setter

Getters and Setters are special methods that allow us to define how to retrieve or set the value of an object property in a clean and controlled way. A getter is a method that gets the value of an object property. It allows us to access a property as usual and (behind the scenes) actually invoking a functio to perform some logic or computation before retrieving the value:

```
const rectangle = {
    width: 10,
    height: 5,

    // Getter for area
    get area() {
      return this.width * this.height;
    }
};

console.log(rectangle.area);  // 50
```

In this example, area is a computed property. Since it is defined using a getter, we can access it like a regular property without parentheses, which helps keep the interface clean.

A setter is a method used to set the value of a property. It allows us to control how a property is updated in a **controlled manner**. It's often used when we want to prevent direct manipulation of a properties, or when setting a value requires additional logic (like validation or transformations):

```
const rectangle = {
    _width: 10,
    _height: 5,

    // Getter for area
    get area() {
      return this._width * this._height;
    },

    // Setter for width
    set width(value) {
      if (value ⩽ 0) {
        console.log("Width must be positive.");
      } else {
```

```
        this._width = value;
      }
    },

    // Setter for height
    set height(value) {
      if (value <= 0) {
        console.log("Height must be positive.");
      } else {
        this._height = value;
      }
    }
};

rectangle.width = 15;  // Valid value, setter is called
rectangle.height = -3;  // Invalid value, setter rejects it
console.log(rectangle.area);  // 75 (after width is updated)
```

In the example, width and height are private properties, but their values are modified through the setters. The setters ensure that the width and height are positive numbers and provide a controlled way to update the values. Notice that the underscore (_) before a property name, like _width or _height, is a **common convention** in JavaScript to indicate that the property is private or intended for internal use only. While JavaScript does not natively support private properties (though this is changing with the introduction of **private fields** in ES2022, bat for classes), the underscore serves as a visual cue to developers that the property should not be accessed or modified directly from outside the object.

1.4.5 Classes

JavaScript in ES 6 introduces also the support for **class-based object-oriented programming**. A class is a blueprint for creating objects, defining properties and methods that all instances of the class will share. Classes provide a more structured and organized way to create objects, similar to classes in other object-oriented languages like Java or C++. A class is defined using the **class** keyword followed by the name. Inside a class, we can define the **constructor** method, which is called when a new instance of the class is created. The constructor is used to initialize the object's properties. We can also define methods that are shared by all instances of the class. These methods can operate on the properties of the instance and define behaviors specific to that object. Here's an example of a class in JavaScript:

```
class Rectangle {
    constructor(width, height) {
      this.width = width;
      this.height = height;
    }

    getArea() {
      return this.width * this.height;
    }
}

const myRectangle = new Rectangle(10, 5);
console.log(myRectangle.getArea());  // 50
```

Classes in JavaScript also support **inheritance**, allowing one class to inherit properties and methods from another class. This is done using the **extends** keyword, which creates a subclass that inherits from a parent class. The subclass can override or extend the functionality of the parent class:

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

    describe() {
      return `This is a shape named ${this.name}`;
    }
}

class Circle extends Shape {
    constructor(name, radius) {
      super(name);  // Calls the parent class's constructor
      this.radius = radius;
    }

    getArea() {
      return Math.PI * this.radius * this.radius;
    }
}

const circle = new Circle('Circle', 10);
console.log(circle.describe());  // "This is a shape named Circle"
```

```
console.log(circle.getArea());   // 314.159
```

Classes also introduce also the concept of **static methods**. These are methods that belong to the class itself, not to instances of the class. Static methods are useful for utility functions or methods that do not depend on instance properties.

```
class MathUtility {
    static add(a, b) {
      return a + b;
    }
}

console.log(MathUtility.add(5, 3));  // 8
```

Classes are essentially **syntactic sugar** over constructor functions, offering a more readable and structured approach to object-oriented programming. While they serve the same purpose as constructor functions, classes provide a more formal and consistent structure. Overall, classes offer a more modern and object-oriented approach, while constructor functions are the older, more flexible method. Despite the differences in syntax and structure, both classes and constructors are used to achieve similar goal, creating objects with shared properties and behaviors. However, classes provide more clarity and support for advanced features like inheritance and method definitions, making them the preferred approach in modern JavaScript.

1.4.6 Polymorphism

Polymorphism is a key concept in object-oriented programming that allows objects of different types to be treated as instances of the same type through a common interface. It describes the ability of different objects to respond to the same method in different ways, depending on their underlying type. Polymorphism provides flexibility and extensibility. It allows developers to write more generalized code that can work with objects of different types, promoting code reuse and reducing redundancy. For example, a function can be written to accept any object that implements a certain method, regardless of the specific class it belongs to. This enables objects of various classes to be treated in a uniform manner, while still allowing each class to implement the method in a way that is appropriate for its own behavior.

```
class Animal {
    speak() {
      console.log("Animal makes a sound");
```

```
    }
}

class Dog extends Animal {
    speak() {
      console.log("Dog barks");
    }
}

const myAnimal = new Animal();
const myDog = new Dog();

myAnimal.speak();  // "Animal makes a sound"
myDog.speak();     // "Dog barks"
```

In this example, both Animal and Dog have a speak method, but the behavior of the method differs depending on the object calling it. This is an example of **method overriding**, where the Dog class provides a new implementation for the speak method.

In JavaScript, polymorphism is not restricted to classes or object-oriented programming. Even without using classes, we can still leverage polymorphism through the use of objects and functions. The concept remains the same: polymorphism allows different types to be treated uniformly by using a common interface, meaning that different objects or functions can exhibit different behaviors while sharing the same method name or function signature. The key is that you can have multiple objects with the same method name, and each object can implement that method differently. This allows us to interact with different objects in a similar manner while enabling them to execute unique behaviors when needed:

```
const dog = {
    speak: function() {
      console.log("Dog barks");
    }
};

const cat = {
    speak: function() {
      console.log("Cat meows");
    }
};

const bird = {
```

```
    speak: function() {
      console.log("Bird chirps");
    }
};

function makeAnimalSpeak(animal) {
    animal.speak();
}

makeAnimalSpeak(dog);   // "Dog barks"
makeAnimalSpeak(cat);   // "Cat meows"
makeAnimalSpeak(bird);  // "Bird chirps"
```

Polymorphism helps achieve **loose coupling** in a system, where components are not tightly dependent on each other. Instead, they rely on a common interface or base class, making the system more flexible and easier to extend. This enables developers to create highly reusable and maintainable code.

1.5 Programming paradigms

We have seen that JavaScript supports multiple programming paradigms, including **procedural**, **functional**, and **object-oriented** programming. Each paradigm offers a **different approach** to structuring code and solving problems, and the language flexibility allows developers to choose the best paradigm for a given task.

1.5.1 Procedural Programming

In procedural programming, the focus is on writing sequences of instructions or procedures (functions) that operate on data. This paradigm is based on the concept of the procedure call, where functions are defined to perform specific tasks and are executed in a step-by-step manner. The program is typically organized around a series of actions or instructions that manipulate data. JavaScript, in its early days, was mainly used in a procedural style, with functions being written to execute particular tasks. For example, a JavaScript program might involve defining functions to handle user input, perform calculations, and update the user interface, all without structuring the program around objects or classes. Let's approach a simple problem (calculating the sum of the squares of even numbers in an array) from the procedural approach:

```
function sumOfEvenSquares(numbers) {
    let sum = 0;

    // Loop through the array
    for (let i = 0; i < numbers.length; i++) {
      // Check if the number is even
      if (numbers[i] % 2 === 0) {
        // Add the square of the even number to the sum
        sum += numbers[i] ** 2;
      }
    }

    return sum;
}

const numbers = [1, 2, 3, 4, 5, 6];
console.log(sumOfEvenSquares(numbers));  // Output: 56
```

In the procedural approach, we use a for loop to iterate through the array, check for even numbers, and calculate their squares. The result is accumulated in the sum variable, which is returned at the end.

1.5.2 Object Oriented Programming

On the other hand, object-oriented programming organizes code around the concept of objects, which are instances of classes. These objects hold both data and methods that define their behaviors. OOP encourages the use of inheritance, encapsulation, and polymorphism to model real-world entities and their interactions. JavaScript adopted OOP principles in ES6 with the introduction of the class syntax, although the language has always allowed object-based programming through the use of object literals and prototypes. Let's rewrite the previous example using an object-oriented approach:

```
class EvenSquaresCalculator {
    constructor(numbers) {
      this.numbers = numbers;
    }

    sumOfEvenSquares() {
      let sum = 0;
```

```
      // Loop through the array of numbers
      for (let number of this.numbers) {
        // Check if the number is even
        if (number % 2 === 0) {
          // Add the square of the even number to the sum
          sum += number ** 2;
        }
      }

      return sum;
    }
}

const calculator = new EvenSquaresCalculator([1, 2, 3, 4, 5, 6]);
console.log(calculator.sumOfEvenSquares());  // Output: 56
```

In the object-oriented style, we encapsulate the logic in an object that represents the process of summing the squares of even numbers. We define a method inside an object to handle the calculation.

1.5.3 Functional Programming

Functional programming, in contrast, emphasizes functions as the primary building blocks of a program. In this paradigm, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions. Functional programming avoids shared state and mutable data, focusing instead on **pure functions** (functions that produce the same output for the same input and have no side effects). JavaScript supports functional programming by allowing anonymous functions, higher-order functions, and closures, making it easier to write code in a functional style. Let's rewrite the previous example using a functional approach:

```
const sumOfEvenSquares = (numbers) =>
    numbers
      .filter(num => num % 2 === 0)    // Get even numbers
      .map(num => num ** 2)             // Square each even number
      .reduce((acc, num) => acc + num, 0); // Sum the squares

  const numbers = [1, 2, 3, 4, 5, 6];
  console.log(sumOfEvenSquares(numbers));  // Output: 56
```

In the functional approach, we use filter to extract the even numbers, map to square them, and reduce to sum the squared values. Each of these operations returns a new array, and the data is processed without mutating any state. The functions are small and composed in a declarative way.

1.5.4 Imperative vs Declarative style

In this context, we can also distinguish between **imperative** and **declarative** programming styles. Imperative programming is about **explicitly describing the steps needed to perform a task**. In this paradigm, the programmer writes a sequence of instructions that tell the computer how to do something. This includes specifying the exact control flow (like loops and conditionals) and managing the state of the program at each step. Imperative code is focused on the control of execution, where the programmer takes on the responsibility of managing the state and providing a step-by-step process. For example, in an imperative style, calculating the sum of even numbers in an array might look like this:

```
let numbers = [1, 2, 3, 4, 5, 6];
let sum = 0;

for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] % 2 === 0) {
    sum += numbers[i];
  }
}

console.log(sum);  // Output: 12
```

Declarative programming, on the other hand, focuses on **describing what we want to achieve without specifying all the steps involved**. In this style, we express the desired result, leaving the underlying mechanism for achieving it abstracted away. Declarative programming abstracts away the control flow and focuses more on the end goal. This approach is generally more readable and concise, as it removes much of the boilerplate code. The same task of summing the even numbers in an array can be written in a declarative style like this:

```
let numbers = [1, 2, 3, 4, 5, 6];
let sum = numbers.filter(num => num % 2 === 0)
                 .reduce((acc, num) => acc + num, 0);

console.log(sum);  // Output: 12
```

In JavaScript, all three paradigms (procedural, OOP and function) can be used together, depending on the needs of the application and the preferences of the developer. JavaScript is particularly well-suited to support these paradigms because it is flexible and allows developers to mix and match approaches within the same application. For example, we might write our data handling and transformations in a functional style, while organizing the logic for user interfaces and complex behaviors in an object-oriented manner. In general, the procedural approach is useful for small, straightforward scripts where we just need to process data in a linear fashion. Object-oriented programming shines in scenarios that require modeling complex entities and behaviors, particularly when dealing with large-scale applications. Functional programming, with its emphasis on immutability and pure functions, is ideal for cases that require high levels of concurrency, easy testing. Of course, both style (imperative and declarative) are often used together. For instance, we can write imperative code for certain low-level tasks that require explicit control, while using declarative methods like map, filter, and reduce for data manipulation. Let's considere an example to comperative the two styles (see *02.declatative_imperative.js*). Suppose to have a JSON string representing epresenting ancestral data. It stores information about individuals, including their name, sex, birth and death years, and their parents' names:

```
var ancetry_file = '[\n  ' + [
    '{"name": "Carolus Haverbeke", "sex": "m", "born": 1832, "died": 1905, "father":
    ↪  "Carel Haverbeke", "mother": "Maria van Brussel"}',
    '{"name": "Emma de Milliano", "sex": "f", "born": 1876, "died": 1956, "father":
    ↪  "Petrus de Milliano", "mother": "Sophia van Damme"}',
    '{"name": "Maria de Rycke", "sex": "f", "born": 1683, "died": 1724, "father":
    ↪  "Frederik de Rycke", "mother": "Laurentia van Vlaenderen"}',
    '{"name": "Jan van Brussel", "sex": "m", "born": 1714, "died": 1748, "father":
    ↪  "Jacobus van Brussel", "mother": "Joanna van Rooten"}',
    ...
    '{"name": "Jacobus Bernardus van Brussel", "sex": "m", "born": 1736, "died":
    ↪  1809, "father": "Jan van Brussel", "mother": "Elisabeth Haverbeke"}'
  ].join(",\n  ") + "\n]";

  var ancestry = JSON.parse(ancetry_file);
```

We want to compute the average age at death for males and females separately, demonstrating a declarative programming style:

```
function plus(a,b){ return a + b; }
function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }
```

```
console.log(ancestry.filter(male).map(age).reduce(plus)/ancestry.filter(male).length);
console.log(ancestry.filter(female).map(age).reduce(plus)/ancestry.filter(female).length);
```

We can adpot a more declarative style by using arrow functions:

```
console.log(ancestry.filter(p => p.sex == "m").map(p => p.died - p.born).reduce((a,b)
↪  => a + b)/ancestry.filter(p => p.sex == "m").length);
```

In contrast, an imperative approach would involve writing a series of loops and conditional statements to achieve the same result:

```
let males = [];
for(let i=0; i<ancestry.length; i++) {
    if(ancestry[i].sex === 'm') males.push(ancestry[i]);
}

let ages = [];
for(let i=0; i<males.length; i++) {
    ages.push(males[i].died - males[i].born);
}

let sum = 0;
for(let i=0; i<ages.length; i++) {
    sum += ages[i];
}

let average = sum / ages.length;
console.log(average);
```

1.6 More advanced topics

JavaScrip is as a versatile and dynamic programming language, it offers several other features and concepts not covered in this section.

Regular expressions (RegEx) are one of the more sophisticated tools available in JavaScript. They allow for pattern-based text searching, validation, and manipulation, making them incredibly useful for tasks such as form validation or extracting specific pieces of data from a string. This powerful feature helps developers write concise code for complex text processing, such as matching email formats, phone numbers, or even parsing log files.

Another fundamental feature of JavaScript is its ability to **interact with the browser** through the Document Object Model (DOM). The DOM provides a structured representation of the HTML page, allowing JavaScript to access and modify the content dynamically. Whether it's adding new elements, changing existing ones, or updating the styling in response to user actions, the DOM provides the interface for JavaScript to shape the webpage in real time, contributing to the dynamic and interactive nature of modern websites.

Event handling is another key feature in JavaScript, particularly when it comes to making webpages interactive. Events are actions triggered by users, such as clicking buttons, typing in input fields, or hovering over elements. JavaScript enables developers to listen for these events and execute specific functions in response. By adding event listeners to elements, developers can manage user interactions in a smooth and seamless way, providing feedback or triggering changes on the page without requiring a full page reload. we will consider this feature in the context of API development using Node.js.

Timers in JavaScript offer additional flexibility by allowing the execution of functions after a specified delay or at regular intervals. This is particularly useful for tasks such as creating animations, auto-refreshing content, or handling timeouts in asynchronous operations.

Finally, **handling forms** in JavaScript adds another layer of interactivity. Forms are essential for collecting user input, and JavaScript allows developers to validate and manipulate that data before it is submitted to the server. This can include checking for empty fields, ensuring proper formatting, or even providing real-time feedback as users complete forms.