Programming JS

Margit Tennosaar

Scope

Scope in JavaScript refers to the accessibility of variables and functions at various parts of your code. It dictates where variables and functions can be accessed or referenced.

- Global Scope: Variables defined outside any function or block are in the global scope and are accessible from anywhere in the code.
- Local (Function) Scope: Variables declared within a function are in the local scope and are only accessible within that function.
- Block Scope (ES6): Introduced in ES6, let and const declarations are block-scoped, meaning they are only accessible within the block they are defined in.

Global scope

```
let globalVar = 'I am a global variable';

function exampleFunction() {
  console.log(globalVar); // Accessible here
}

console.log(globalVar); // Also accessible here
```

Local scope

```
function exampleFunction() {
  let localVar = "I am a local variable";
  console.log(localVar); // Accessible here
}

console.log(localVar); // ❌ Uncaught ReferenceError: localVar is not defined
```

Local scope

```
let localVar = 'I am THE local variable’;

function exampleFunction1() {
  localVar = 'I am a local variable one’;
  console.log(localVar); // Accessible here
}

function exampleFunction2() {
  console.log(localVar); // Accessible here
  localVar = 'I am a local variable two’;
  console.log(localVar); // Accessible here
}
console.log(localVar)
exampleFunction2();
exampleFunction1();
console.log(localVar)
```

Block scope

```
if (true) {
  let blockVar = 'I am a block-scoped variable';
  console.log(blockVar); // Accessible here
}

console.log(blockVar); // Uncaught ReferenceError: blockVar is not defined
```

Best practices

- Prefer let and const over var to avoid unintended consequences of hoisting.
- Declare variables at the top of their scope for clarity.
- Keep global variables to a minimum to avoid cluttering the global namespace.

Common pitfalls

- Not understanding the difference between var, let, and const.
- Accidentally overwriting global variables due to scope misunderstandings.
- Assuming that block-scoped variables (with let and const) will behave like function-scoped (var) variables.

```
console.log(myLetVar); // ReferenceError: Cannot access 'myLetVar' before initialization
let myLetVar = 5;

console.log(myConstVar); // ReferenceError: Cannot access 'myConstVar' before initialization
const myConstVar = 10;
```

Arrays

Arrays in JavaScript are powerful and flexible structures for storing ordered collections of data. Arrays are **zero-indexed** (first element is at index 0).

```
const array_name = [item1, item2, ...];
```

Creating arrays

```
const emptyArray = []; // Empty array
const numberArray = [1, 2, 3, 4]; // Numbers
const stringArray = ["a", "b", "c"]; // Strings
const mixedArray = [1, "a", true, null]; // Mixed values
```

Adding/Removing Elements

```
let fruits = ["apple", "banana"];
fruits.push("orange");  // Add to end
fruits.pop();           // Remove last item

fruits.unshift("mango"); // Add to beginning
fruits.shift();         // Remove first item
```

- push() and pop() affect the end of an array.
- unshift() and shift() affect the beginning.

Finding elements in array

```
const numbers = [10, 20, 30, 40];

console.log(numbers.indexOf(20)); // 1
console.log(numbers.includes(50)); // false
```

- indexOf() returns the first occurrence of a value (or -1 if not found).
- includes() checks if an element exists (true/false).

Looping through arrays

```
const colors = ["red", "blue", "green"];

colors.forEach(color => console.log(color)); // Prints each color
```

- No need for for loops
- Runs the callback function for each element

Looping through arrays (easier syntax)

```
const colors = ["red", "blue", "green"];

for (const color of colors) {
   console.log(color); // Prints each color
}
```

- Simpler than forEach() when you don’t need an index.
- Works on any iterable (arrays, strings, NodeLists).

Transforming array

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);

console.log(doubled); // [2, 4, 6]
```

- **map() returns a new array with modified values.**

Filtering arrays

```
const ages = [12, 18, 25, 30];

const adults = ages.filter(age => age >= 18);
console.log(adults); // [18, 25, 30]
```

- **filter() returns only matching elements**

- **Use filter() when you need to extract specific elements.**

Finding an element

```
const users = [
   { name: "Alice", age: 25 },
   { name: "Bob", age: 17 },
];

const firstAdult = users.find(user => user.age >= 18);
console.log(firstAdult); // { name: "Alice", age: 25 }
```

- **find() stops at the first match**, while filter() returns all matches.

Combining arrays

```
const arr1 = [1, 2];
const arr2 = [3, 4];

const combined = arr1.concat(arr2);
console.log(combined); // [1, 2, 3, 4]
```

- **Merge arrays using concat()**

- **Alternative:** Using the spread operator (...)

```
const merged = [...arr1, ...arr2];
```

Removing and slicing

```
let items = ["a", "b", "c"];
items.splice(1, 1); // Removes "b"
console.log(items); // ["a", "c"]
```

- Remove elements with splice()
- **Use slice() to create a copy of an array**

```
const copy = items.slice(0, 2);
console.log(copy); // ["a", "b"]
```

- **Best Practice:** Prefer slice() over splice() when **you don’t want to modify the original array.**

Sorting and reversing

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

numbers.sort(); // [1, 2, 3, 5]
numbers.reverse(); // [5, 3, 2, 1]
```

By default, sort() sorts alphabetically, even for numbers!

```
numbers.sort((a, b) => a - b); // Correct ascending order
```

Checking conditions in array

```
const scores = [85, 90, 78, 92];

console.log(scores.some(score => score < 80)); // true (at least one <80)
console.log(scores.every(score => score > 70)); // true (all >70)
```

- some() checks if **at least one** item meets the condition.

  every() checks if **all** items meet the condition.

Reducing arrays

```
const numbers = [10, 20, 30];

const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // 60
```

- reduce() calculates a single value from an array

Array summary

- Arrays store multiple values in a single variable.
- Methods to modify arrays: push(), pop(), shift(), splice(), concat().
- Looping: forEach(), for...of.
- Transforming: map(), filter(), find().
- Checking conditions: some(), every().

Callbacks

Callback function

A callback function is a function that is **passed as an argument** to another function and **executed later.**

```
function greet(name) {
   console.log(`Hello, ${name}!`);
}

function processUser(callback) {
   const user = "Alice";
   callback(user);
}

processUser(greet); // Calls greet("Alice"), outputs: "Hello, Alice!"
```

Callback in arrow function

```
function processUser(callback) {
    const user = "Alice";
    callback(user);
}

// Using an arrow function as a callback
processUser(user => console.log(`Hello, ${user}!`));
```

Why to use callbacks?

- Cleaner, reusable code
- Keeps logic inside the function call
- Avoids writing manual loops

Callbacks in forEach()

forEach() uses callbacks to apply a function to each item in an array.

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

numbers.forEach(function(number) {
   console.log(number * 2);
});



numbers.forEach(num => console.log(num * 2));
```

Most common callbacks

Callback Usage	What It Does
forEach()	Loops through an array
map()	Transforms each item in an array
filter()	Filters array items based on a condition
find()	Finds the first matching item
setTimeout()	Delays execution
setInterval()	Runs repeatedly at a set interval
addEventListener()	Handles user interactions
.then() (Promises)	Handles async operations