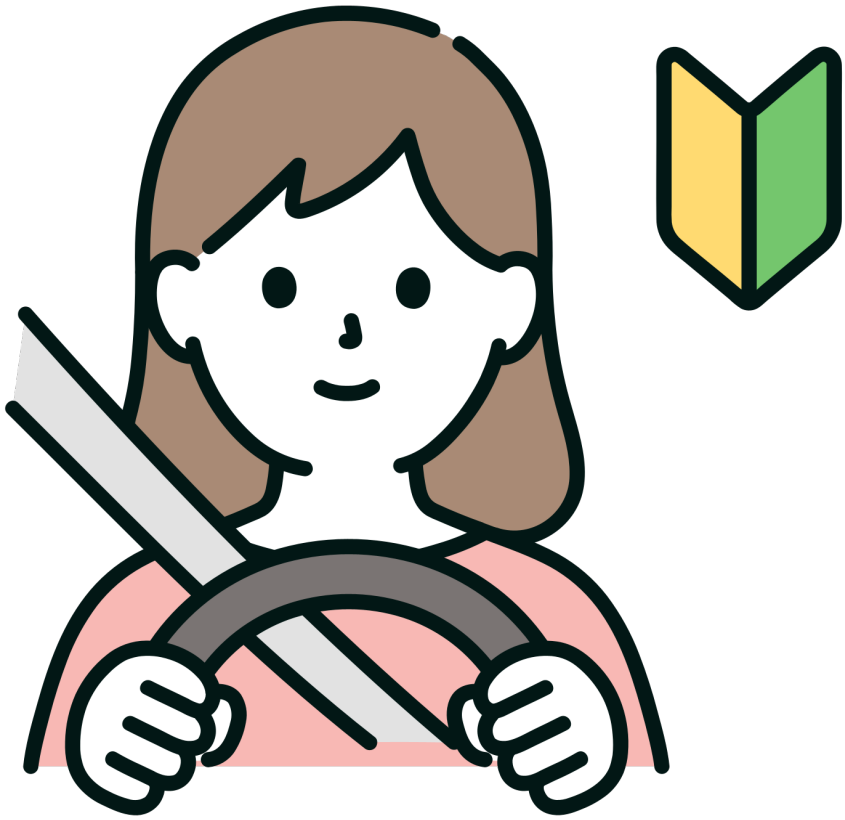
JavaScript for Beginners



readbytes

JavaScript for Beginners

From Basics to Advanced Programming

readbytes.github.io

2025-07-14

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to JavaScript

1. What is JavaScript?
2. Where JavaScript Runs (Browser, Server, Desktop, etc.)
3. Setting Up Your Development Environment (VS Code, Browsers, Node.js)

1 Introduction to JavaScript

1.1 What is JavaScript?

JavaScript is a powerful programming language primarily used to make websites interactive and dynamic. If you think of a website as a building, HTML is the structure—the walls, floors, and roof that create the shape and layout. CSS is the paint and decoration that makes the building look appealing by adding colors, fonts, and styles. JavaScript, then, is the electrical system and appliances that bring the building to life—turning on lights, running elevators, or playing music when you enter a room.

Unlike HTML and CSS, which are declarative languages focused on structure and style, JavaScript is a full-fledged programming language. This means it can perform logical operations, make decisions, and respond to user actions. For example, when you click a button to submit a form, JavaScript checks if you filled it out correctly before sending the data. It can also change content on the fly, create animations, or even communicate with servers to update data without refreshing the page.

JavaScript runs primarily in your web browser, allowing web pages to respond immediately to user input. This real-time interaction is essential for modern web applications like social media platforms, online shopping carts, or games played directly in the browser.

Because JavaScript is a programming language, it supports fundamental concepts like variables (to store data), functions (to perform tasks), and control flow (to decide what happens next). This flexibility has led to its use beyond browsers—for example, on servers with Node.js or even in mobile apps.

1.2 Where JavaScript Runs (Browser, Server, Desktop, etc.)

JavaScript is a versatile language that runs in many different environments, making it one of the most widely used programming languages today. Originally designed to run inside web browsers, JavaScript's capabilities have expanded far beyond the browser window.

In the Browser: JavaScript's primary environment is the web browser. Every modern browser—such as Chrome, Firefox, Safari, and Edge—includes a JavaScript engine that executes scripts on web pages. This allows developers to create interactive websites where users can click buttons, submit forms, play games, or see real-time updates without refreshing the page. For example, social media platforms like Facebook and Twitter rely heavily on JavaScript to provide dynamic content and smooth user experiences.

On the Server with Node.js: Node.js is a popular runtime environment that lets you run JavaScript code outside of a browser, on servers. This means you can build full web applications entirely with JavaScript, handling both the front-end (user interface) and back-end (server logic). For example, many websites and APIs use Node.js to manage data,

authenticate users, or connect to databases.

Mobile Development: JavaScript is also used to build mobile applications. Frameworks like React Native and Ionic allow developers to write JavaScript code that runs on both iOS and Android devices. This approach enables building apps that work across platforms without rewriting code in different languages like Swift or Java.

Desktop Applications: JavaScript can even power desktop applications through frameworks like Electron. Apps like Visual Studio Code, Slack, and Discord are built using JavaScript combined with web technologies, giving developers the ability to create cross-platform desktop software using familiar tools.

1.3 Setting Up Your Development Environment (VS Code, Browsers, Node.js)

To start writing and running JavaScript effectively, it's important to set up a development environment with the right tools. In this section, we'll walk through setting up Visual Studio Code (VS Code), installing Node.js, and using your browser's developer console.

1. Installing Visual Studio Code

Visual Studio Code is a free, lightweight, and powerful code editor widely used by developers. It offers features like syntax highlighting, debugging, and extensions that make coding easier.

- Visit the VS Code website.
- Click the **Download** button for your operating system (Windows, macOS, or Linux).
- Run the installer and follow the prompts to complete the installation.

Once installed, open VS Code and create a new file with a `.js` extension to start writing JavaScript code. VS Code provides helpful features like auto-completion and error highlighting.

2. Installing Node.js

Node.js allows you to run JavaScript outside the browser, directly on your computer, which is essential for server-side development and testing.

- Go to the Node.js download page.
- Download the recommended LTS (Long-Term Support) version.
- Run the installer and follow the setup steps.

To verify the installation, open your terminal (Command Prompt on Windows, Terminal on macOS/Linux) and type:

```
node -v
```

You should see the Node.js version number printed. This means Node.js is ready to use.

You can now run JavaScript files by navigating to their folder in the terminal and typing:

```
node filename.js
```

3. Using the Browser Developer Console

Modern browsers come with built-in developer tools that let you write and test JavaScript instantly.

- Open your preferred browser (Chrome, Firefox, Edge, or Safari).
- Press `F12` or right-click on a webpage and select **Inspect**.
- Click on the **Console** tab.

In the console, you can type JavaScript commands and see the results immediately. For example:

```
console.log('Hello, World!');
```

This is perfect for experimenting with code snippets or debugging your scripts.

Summary

- **VS Code** is your main editor for writing JavaScript files with smart features.
- **Node.js** lets you run JavaScript on your computer outside the browser.
- **Browser consoles** provide quick ways to test and debug JavaScript live.

With these tools set up, you're ready to start learning and building JavaScript programs!

Chapter 2.

Your First JavaScript Program

1. Hello World in JavaScript
2. Using `<script>` in HTML
3. Running JavaScript in the Browser Console
4. Commenting Your Code

2 Your First JavaScript Program

2.1 Hello World in JavaScript

The classic first step in learning any programming language is creating a program that displays the message "Hello, World!" JavaScript makes this easy and fun.

Using the Browser Console

You can run JavaScript directly in your browser's console without creating any files. To open the console:

- In Chrome or Firefox, press `F12` or right-click anywhere on the page and select **Inspect**, then click the **Console** tab.

Once open, type the following code and press Enter:

```
console.log("Hello, World!");
```

This command tells the browser to print "Hello, World!" in the console. You should see the message appear immediately.

Using `<script>` in an HTML Page

To embed JavaScript in an HTML page, you use the `<script>` tag. Here is a simple example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello World Example</title>
</head>
<body>
  <script>
    console.log("Hello, World!");
    alert("Hello, World!");
  </script>
</body>
</html>
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello World Example</title>
</head>
<body>
  <script>
    console.log("Hello, World!");
    alert("Hello, World!");
  </script>
</body>
</html>
```

Save this code as `hello.html` and open it in your browser. The message "Hello, World!" will

be logged to the console, and an alert popup will appear on the screen.

This simple example demonstrates how JavaScript can output messages both invisibly (console) and visibly (alert), helping you get started with writing and running your own code.

2.2 Using <script> in HTML

JavaScript code is added to an HTML document using the <script> tag. This tag tells the browser to execute the JavaScript either directly within the HTML file (inline) or by loading an external JavaScript file.

Inline JavaScript

You can write JavaScript code directly inside the <script> tag in your HTML file. For example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline Script Example</title>
  <script>
    console.log("This is inline JavaScript in the head.");
  </script>
</head>
<body>
  <h1>Hello from Inline JavaScript!</h1>
  <script>
    alert("Hello from inline JavaScript in the body!");
  </script>
</body>
</html>
```

Full runnable code:

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline Script Example</title>
  <script>
    console.log("This is inline JavaScript in the head.");
  </script>
</head>
<body>
  <h1>Hello from Inline JavaScript!</h1>
  <script>
    alert("Hello from inline JavaScript in the body!");
  </script>
</body>
</html>
```

In this example, there are two inline scripts: one inside the <head> and one inside the <body>. Both run when the browser parses their location in the HTML.

External JavaScript Files

For larger scripts or to keep your HTML clean, JavaScript code is often stored in separate `.js` files and linked using the `src` attribute:

```
<!DOCTYPE html>
<html>
<head>
  <title>External Script Example</title>
  <script src="script.js"></script>
</head>
<body>
  <h1>Using External JavaScript</h1>
</body>
</html>
```

The file `script.js` might contain:

```
console.log("This is code from an external file.");
alert("External script loaded!");
```

Where to Place the `<script>` Tag

- **In the `<head>`:** Scripts here load before the page content. This can delay the page rendering, especially if the script is large or slow to load.
- **Before the closing `</body>` tag:** Placing scripts here lets the browser load and display the HTML content first, improving page load speed and user experience. This is the preferred practice for most scripts that manipulate page content.

In summary, you can include JavaScript inline within your HTML or link to external files. Placement of the `<script>` tag affects how and when your JavaScript runs during page loading.

2.3 Running JavaScript in the Browser Console

The browser console is a powerful tool that lets you write and run JavaScript code instantly without needing to create any files. It's perfect for experimenting, testing snippets, and debugging.

Opening the Developer Console

Here's how to open the console in popular browsers:

- **Google Chrome:** Press `Ctrl + Shift + J` (Windows/Linux) or `Cmd + Option + J` (Mac). Alternatively, right-click anywhere on the page and select **Inspect**, then click the **Console** tab.
- **Mozilla Firefox:** Press `Ctrl + Shift + K` (Windows/Linux) or `Cmd + Option + K` (Mac). Or right-click and choose **Inspect Element**, then select **Console**.

- **Microsoft Edge:** Press `Ctrl + Shift + J` (Windows) or `Cmd + Option + J` (Mac). Right-click and choose **Inspect**, then go to the **Console** tab.

Using the Console

Once the console is open, you'll see a prompt where you can type JavaScript code. Press Enter to run your code and see the results immediately.

Try these examples:

```
2 + 3
```

This will output `5`—simple math in action!

```
console.log("Hello from the console!");
```

This prints the message in the console log.

```
"JavaScript".toUpperCase()
```

This converts the string to uppercase and shows `"JAVASCRIPT"`.

The console is an excellent way to practice JavaScript commands quickly and see how they work. It also helps you debug errors by showing messages and letting you inspect variables on live web pages. Get comfortable with it—it's one of the best tools for learning and developing in JavaScript!

2.4 Commenting Your Code

Comments are lines in your JavaScript code that the browser ignores when running the program. They are used to explain what the code does, making it easier to understand for yourself and others who read your code later.

There are two types of comments in JavaScript:

Single-line comments start with `//` and continue until the end of the line. Use these for short explanations:

```
// This logs a greeting message to the console
console.log("Hello, World!");
```

Multi-line comments begin with `/*` and end with `*/`. They can span several lines, making them useful for longer notes or temporarily disabling blocks of code:

```
/*
 This function adds two numbers
 and returns the result.
*/
function add(a, b) {
  return a + b;
}
```

Why Comments Matter

Comments help make your code more readable and maintainable. When you revisit your code after days or share it with others, comments explain your logic and purpose, saving time and reducing errors. For beginners, writing comments encourages thinking clearly about what each part of the code does.

Good Commenting Practices

- Keep comments clear and concise.
- Avoid obvious comments that repeat what the code does.
- Use comments to explain *why* you wrote the code, not just *what* it does.

Example:

```
// Calculate area of a circle using radius
const radius = 5;
const area = Math.PI * radius * radius; // Pir² formula
console.log(area);
```

By practicing commenting, your code becomes easier to understand and debug!

Chapter 3.

Variables and Data Types

1. Declaring Variables: `var`, `let`, `const`
2. Primitive Types: Number, String, Boolean, Undefined, Null, Symbol, BigInt
3. Type Checking with `typeof`

3 Variables and Data Types

3.1 Declaring Variables: `var`, `let`, `const`

In JavaScript, variables are containers for storing data values. You can declare variables using three keywords: `var`, `let`, and `const`. Understanding their differences is important for writing clean, bug-free code.

`var`

`var` is the oldest way to declare variables and has some quirks:

- **Function scope:** Variables declared with `var` are scoped to the nearest function, not blocks like `if` or loops.
- **Hoisting:** `var` declarations are hoisted to the top of their scope, meaning you can use the variable before it's declared (though its value will be `undefined` until assignment).

Example:

Full runnable code:

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

if (true) {
  var y = 10;
}
console.log(y); // 10 (y is accessible outside the if-block)
```

`let`

`let` was introduced in ES6 to fix `var`'s scoping problems:

- **Block scope:** Variables declared with `let` exist only inside the nearest `{}` block.
- **No hoisting (temporal dead zone):** You cannot access a `let` variable before it's declared.

Example:

```
if (true) {
  let a = 20;
  console.log(a); // 20
}
// console.log(a); // Error: a is not defined
```

`let` variables can be reassigned:

```
let count = 1;
count = 2; // valid
```

`const`

`const` declares constants—variables that cannot be reassigned after initialization:

- **Block scoped**, like `let`.
- Must be initialized when declared.
- The reference cannot change, but objects assigned to `const` can have their properties modified.

Example:

```
const PI = 3.14;
// PI = 3.15; // Error: Assignment to constant variable

const obj = { name: "Alice" };
obj.name = "Bob"; // Allowed: modifying property, not reassigning obj
```

Best Practices

- Prefer `const` by default for values that don't change.
- Use `let` when you expect to reassign the variable.
- Avoid `var` to reduce bugs related to hoisting and scope confusion.

Using `let` and `const` leads to more predictable and readable code, which is especially helpful for beginners.

3.2 Primitive Types: Number, String, Boolean, Undefined, Null, Symbol, BigInt

JavaScript has seven primitive data types. Primitives are the most basic kinds of data and are immutable, meaning their values cannot be changed once created. Understanding these types is fundamental to writing effective JavaScript.

1. Number

The `Number` type represents all numeric values, including integers and floating-point numbers.

```
let age = 25;
let price = 19.99;
```

Note: JavaScript uses double-precision floating-point format, which can cause small rounding errors:

```
console.log(0.1 + 0.2); // 0.30000000000000004
```

This can confuse beginners, so be cautious with floating-point math.

2. String

Strings represent text and are enclosed in single quotes `'string'`, double quotes `"string"`, or backticks ‘ (for template literals).

```
let name = "Alice";
let greeting = 'Hello';
let message = `Welcome, ${name}!`;
```

Strings are immutable — you cannot change a character directly but can create new strings from existing ones.

3. Boolean

Boolean values are either `true` or `false`, often used in conditions:

```
let isLoggedIn = true;
let hasAccess = false;
```

4. Undefined

A variable that has been declared but not assigned a value is `undefined`:

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

This means "no value has been assigned yet."

5. Null

`null` is a special value that represents "no value" or "empty." It is explicitly assigned to indicate the absence of any object value.

```
let user = null;
```

Common pitfall: `null` and `undefined` are different but often confused. `undefined` means a variable hasn't been assigned, while `null` means it has been intentionally cleared.

6. Symbol

Symbols are unique and immutable identifiers, often used to create unique object keys:

```
let id = Symbol("id");
```

Every symbol is unique, even if they have the same description.

7. BigInt

`BigInt` allows representation of integers larger than the safe limit for `Number` (`2^53 - 1`):

```
let bigNumber = 9007199254740991n;
```

BigInts are created by appending `n` to the end of an integer.

Summary

Primitive types are the building blocks of JavaScript data. Be mindful of differences like `null` vs. `undefined` and floating-point quirks with numbers. Knowing these types well will help you avoid common mistakes and write better code.

3.3 Type Checking with `typeof`

In JavaScript, the `typeof` operator is used to determine the type of a variable or value. It's a simple yet powerful tool to help you understand what kind of data you're working with.

The syntax is straightforward:

```
typeof value
```

Here are examples using all the primitive types:

Full runnable code:

```
console.log(typeof 42);            // "number"
console.log(typeof "Hello");       // "string"
console.log(typeof true);          // "boolean"
console.log(typeof undefined);     // "undefined"
console.log(typeof Symbol("id")); // "symbol"
console.log(typeof 9007199254740991n); // "bigint"
console.log(typeof null);          // "object"
```

Most of these are intuitive, but note the special case with `null`. Even though `null` represents the absence of a value, `typeof null` returns `"object"`. This is a long-standing quirk in JavaScript, originating from its early implementation, and has been kept for backward compatibility. So, always remember that `null` is a primitive type, but `typeof` incorrectly labels it as an object.

Using `typeof` with Variables

Full runnable code:

```
let name = "Alice";
console.log(typeof name); // "string"

let count;
console.log(typeof count); // "undefined"

let isActive = false;
console.log(typeof isActive); // "boolean"
```

Why Use `typeof`?

`typeof` is useful when writing conditional code that depends on data types, such as validating inputs or debugging your programs. While it doesn't distinguish between objects like arrays or dates, it is reliable for checking primitive types.

Understanding how `typeof` works helps you write clearer, more robust JavaScript programs.

Chapter 4.

Operators and Expressions

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. String Concatenation
6. Operator Precedence

4 Operators and Expressions

4.1 Arithmetic Operators

Arithmetic operators in JavaScript allow you to perform basic mathematical operations on numbers. These operators are essential for calculations in everyday programming tasks like computing totals, taxes, or discounts.

1. Addition (+)

Adds two numbers or combines strings.

```
let price = 20 + 5;           // 25
let greeting = "Hello, " + "World!"; // "Hello, World!"
```

Analogy: Adding items in your shopping cart to get the total price.

2. Subtraction (-)

Subtracts one number from another.

```
let balance = 100 - 30;       // 70
let difference = 50 - 20;     // 30
```

Analogy: Calculating how much money remains after a purchase.

3. Multiplication (*)

Multiplies two numbers.

```
let totalCost = 10 * 3;       // 30
let area = 5 * 4;             // 20
```

Analogy: Finding the total price when buying multiple items.

4. Division (/)

Divides one number by another.

```
let pricePerItem = 30 / 3;    // 10
let average = 100 / 4;        // 25
```

Analogy: Splitting a bill evenly among friends.

5. Modulus (%)

Returns the remainder after division.

```
console.log(10 % 3);          // 1
console.log(15 % 5);          // 0
```

Analogy: Checking if a number is even or odd by seeing if the remainder is 0 or not.

6. Exponentiation (**)

Raises a number to the power of another.

```
let squared = 4 ** 2;         // 16
let cubed = 3 ** 3;           // 27
```

Analogy: Calculating area or volume when dimensions are squared or cubed.

Full runnable code:

```
// 1. Addition
let price = 20 + 5;
let greeting = "Hello, " + "World!";
console.log("Addition:");
console.log("price:", price);                // 25
console.log("greeting:", greeting);          // "Hello, World!"

// 2. Subtraction
let balance = 100 - 30;
let difference = 50 - 20;
console.log("Subtraction:");
console.log("balance:", balance);            // 70
console.log("difference:", difference);      // 30

// 3. Multiplication
let totalCost = 10 * 3;
let area = 5 * 4;
console.log("Multiplication:");
console.log("totalCost:", totalCost);        // 30
console.log("area:", area);                  // 20

// 4. Division
let pricePerItem = 30 / 3;
let average = 100 / 4;
console.log("Division:");
console.log("pricePerItem:", pricePerItem); // 10
console.log("average:", average);            // 25

// 5. Modulus
console.log("Modulus:");
console.log("10 % 3 =", 10 % 3);            // 1
console.log("15 % 5 =", 15 % 5);            // 0

// 6. Exponentiation
let squared = 4 ** 2;
let cubed = 3 ** 3;
console.log("Exponentiation:");
console.log("squared:", squared);            // 16
console.log("cubed:", cubed);                // 27
```

These operators form the foundation for all math in JavaScript, helping you solve real-world problems involving numbers quickly and clearly.

4.2 Assignment Operators

Assignment operators in JavaScript are used to assign values to variables. The most basic assignment operator is =, but there are also compound operators that combine arithmetic operations with assignment, making your code shorter and clearer.

1. Simple Assignment (=)

Assigns the value on the right to the variable on the left.

Full runnable code:

```
let total = 10; // total is now 10
console.log(total);
```

2. Addition Assignment (+=)

Adds the value on the right to the variable, then assigns the result back to the variable.

Full runnable code:

```
let total = 10;
total += 5; // equivalent to total = total + 5
console.log(total); // 15
```

Use this to increase a variable by a certain amount without rewriting the whole expression.

3. Subtraction Assignment (-=)

Subtracts the value on the right from the variable.

Full runnable code:

```
let balance = 50;
balance -= 20; // equivalent to balance = balance - 20
console.log(balance); // 30
```

4. Multiplication Assignment (*=)

Multiplies the variable by the value on the right.

Full runnable code:

```
let price = 10;
price *= 3; // equivalent to price = price * 3
console.log(price); // 30
```

5. Division Assignment (/=)

Divides the variable by the value on the right.

Full runnable code:

```
let total = 30;
total /= 5; // equivalent to total = total / 5
console.log(total); // 6
```

6. Modulus Assignment (%=)

Calculates the remainder of the variable divided by the value on the right.

Full runnable code:

```
let count = 10;
count %= 3; // equivalent to count = count % 3
console.log(count); // 1
```

Why Use Assignment Operators?

These compound operators help simplify and shorten your code, making it more readable and efficient. Instead of writing `x = x + 10`, you can write `x += 10`. This also reduces errors and makes updating variable values quicker and clearer. Use them whenever you want to update a variable based on its current value!

4.3 Comparison Operators

Comparison operators in JavaScript are used to compare two values. These comparisons return a Boolean value — either `true` or `false` — which is essential for decision-making in your programs, such as in `if` statements and loops.

1. Equality (==)

Checks if two values are equal, **with type coercion**. This means JavaScript converts types if needed before comparing.

Full runnable code:

```
console.log(5 == '5');  // true (string '5' is converted to number 5)
console.log(0 == false); // true (false is converted to 0)
```

Because `==` converts types, it can sometimes give unexpected results. Use it carefully.

2. Strict Equality (===)

Checks if two values are equal **without type coercion** — both value and type must match.

Full runnable code:

```
console.log(5 === '5');  // false (number vs string)
console.log(5 === 5);    // true
```

Use `===` for more predictable and safer comparisons.

3. Inequality (!=)

Checks if two values are **not equal**, with type coercion.

Full runnable code:

```
console.log(5 != '10'); // true
console.log(5 != 5);    // false
```

4. Strict Inequality (!==)

Checks if two values are **not equal** without type coercion.

Full runnable code:

```
console.log(5 !== '5'); // true
console.log(5 !== 5);   // false
```

5. Greater Than (>) and Less Than (<)

Compare if one value is larger or smaller than the other.

Full runnable code:

```
console.log(10 > 5);    // true
console.log(3 < 2);     // false
console.log('apple' > 'banana'); // false (string comparison based on Unicode)
```

6. Greater Than or Equal To (>=) and Less Than or Equal To (<=)

Check if a value is greater/less than or equal to another.

Full runnable code:

```
console.log(5 >= 5);    // true
console.log(4 <= 3);    // false
```

Why Comparison Operators Matter

They control program flow by enabling conditions like:

Full runnable code:

```
let age = 30;
if (age >= 18) {
  console.log("You can vote.");
} else {
  console.log("You are too young.");
}
```

Summary

- Use === and !== for strict comparisons (recommended).
- Understand that == and != perform type coercion, which can lead to unexpected results.

- Comparison operators are fundamental in making decisions and controlling the logic in your programs.

4.4 Logical Operators

Logical operators in JavaScript allow you to combine or modify Boolean values (`true` or `false`). They are essential when you want to check multiple conditions at once or invert a condition in your code, especially inside `if` statements.

1. AND (`&&`)

The `&&` operator returns `true` only if **both** conditions are true. Otherwise, it returns `false`.

Full runnable code:

```
let isLoggedIn = true;
let hasPermission = true;

if (isLoggedIn && hasPermission) {
  console.log("Access granted");
} else {
  console.log("Access denied");
}
```

Analogy: You need both a ticket **and** an ID to enter a concert.

2. OR (`||`)

The `||` operator returns `true` if **at least one** of the conditions is true. It only returns `false` if **both** are false.

Full runnable code:

```
let isAdmin = false;
let isModerator = true;

if (isAdmin || isModerator) {
  console.log("You can edit posts");
} else {
  console.log("You cannot edit posts");
}
```

Analogy: You can enter if you have a membership card **or** a guest pass.

3. NOT (`!`)

The `!` operator inverts a Boolean value: it turns `true` into `false`, and vice versa.

Full runnable code:

```
let isGuest = false;

if (!isGuest) {
  console.log("Welcome back, user!");
}
```

This means "if **not** a guest, show welcome message."

Practical Example: Form Validation

Full runnable code:

```
let username = "user123";
let password = "pass";

if (username !== "" && password !== "") {
  console.log("Form submitted");
} else {
  console.log("Please fill in all fields");
}
```

Here, the form only submits if **both** username and password are filled.

Summary

Logical operators help combine multiple conditions (`&&`, `||`) or reverse a condition (`!`). They are fundamental in controlling program flow and building complex decision-making in your JavaScript code.

4.5 String Concatenation

String concatenation means joining two or more strings together to form one combined string. In JavaScript, you can do this in two main ways: using the `+` operator or using template literals.

1. Using the + Operator

The `+` operator joins strings simply by placing them side by side.

Full runnable code:

```
let firstName = "Alice";
let lastName = "Johnson";

let fullName = firstName + " " + lastName;
console.log(fullName); // Output: Alice Johnson
```

Here, we add a space `" "` between the first and last names to separate them.

2. Using Template Literals

Template literals (introduced in ES6) allow embedding variables directly inside strings using backticks and${}‘ placeholders.

Full runnable code:

```
let firstName = "Alice";
let lastName = "Johnson";

let fullName = `${firstName} ${lastName}`;
console.log(fullName); // Output: Alice Johnson
```

Why Prefer Template Literals?

- Cleaner syntax without the need for manual spaces or + operators.
- Easier to read and write, especially with multiple variables.
- Supports multi-line strings without special characters.

Mini Activity: Create a Greeting Message

Try this yourself:

Full runnable code:

```
let userName = "Bob";
let day = "Monday";

let greeting = `Hello, ${userName}! Happy ${day}!`;
console.log(greeting); // Output: Hello, Bob! Happy Monday!
```

Summary

While the + operator works well for simple concatenation, template literals make combining strings with variables easier and more readable. Start using template literals to write cleaner, modern JavaScript code!

4.6 Operator Precedence

Operator precedence determines the order in which JavaScript evaluates parts of an expression when multiple operators are used. Just like in math, some operations happen before others unless you use parentheses to change the order.

How JavaScript Evaluates Expressions

For example, multiplication (*) has higher precedence than addition (+), so it is evaluated first:

Full runnable code:

```
let result = 3 + 4 * 5;
console.log(result); // Outputs 23, not 35
```

Here, `4 * 5` is calculated first (equals 20), then added to 3.

Using Parentheses to Control Order

You can use parentheses `()` to force certain operations to happen first:

Full runnable code:

```
let result = (3 + 4) * 5;
console.log(result); // Outputs 35
```

Now, `3 + 4` is calculated first (equals 7), then multiplied by 5.

Operator Precedence in Logical Expressions

Logical operators also have precedence. The NOT operator (`!`) has higher precedence than AND (`&&`) and OR (`||`):

Full runnable code:

```
let a = true;
let b = false;

console.log(!a && b); // false, because !a is false, then false && false is false
console.log(!(a && b)); // true, parentheses change evaluation order
```

Why Operator Precedence Matters

Understanding operator precedence helps you predict how complex expressions are evaluated and avoid bugs. When in doubt, use parentheses—they make your code clearer and ensure it runs as intended.

Summary Table (Simplified)

Precedence Level	Operator	Example
High	`()`, `!`	`(2 + 3) * 4`
Medium	`*`, `/`, `%`	`4 * 5 + 2`
Low	`+`, `-`	`3 + 4 * 5`
Lowest	`&&`, `||`	`true && false`

Use operator precedence knowledge to write accurate and readable JavaScript expressions!

Chapter 5.

Control Flow and Decision Making

1. `if`, `else`, and `else if`
2. `switch` Statements
3. Ternary Operator

5 Control Flow and Decision Making

5.1 `if`, `else`, and `else if`

Conditional statements allow your JavaScript program to make decisions and run different code depending on whether certain conditions are true or false. The most common way to do this is with `if`, `else if`, and `else` blocks.

Basic `if` Statement

An `if` statement runs a block of code only if the condition inside the parentheses is true:

Full runnable code:

```
let age = 18;

if (age >= 18) {
  console.log("You are an adult.");
}
```

If `age` is 18 or more, the message appears; otherwise, nothing happens.

Adding an `else`

Use `else` to run code when the condition is false:

Full runnable code:

```
let age = 15;

if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}
```

Using `else if` for Multiple Conditions

`else if` lets you check multiple conditions in sequence:

Full runnable code:

```
let temperature = 30;

if (temperature > 35) {
  console.log("It's very hot!");
} else if (temperature > 20) {
  console.log("It's warm.");
} else {
  console.log("It's cold.");
}
```

Nested Conditions

You can nest `if` statements inside each other to create more complex logic:

Full runnable code:

```
let userRole = "admin";
let isLoggedIn = true;

if (isLoggedIn) {
  if (userRole === "admin") {
    console.log("Welcome, admin!");
  } else {
    console.log("Welcome, user!");
  }
} else {
  console.log("Please log in.");
}
```

Combining Conditions with Logical Operators

Use `&&` (AND) and `||` (OR) to combine conditions:

Full runnable code:

```
let age = 22;
let hasID = true;

if (age >= 18 && hasID) {
  console.log("Entry allowed.");
} else {
  console.log("Entry denied.");
}
```

Summary

- Use `if` to run code when a condition is true.
- Use `else` to handle the false case.
- Use `else if` to check multiple conditions.
- Nest and combine conditions for more control.

Mastering these conditional statements helps your program make decisions based on data and user input.

5.2 `switch` Statements

The `switch` statement offers a clean and organized way to compare a value against many possible cases. It's often used as an alternative to multiple `if-else` blocks, especially when you're checking the same variable against different values.

How `switch` Works

- The value inside the `switch(expression)` is compared to each `case` value.

- When a match is found, the code under that `case` runs.
- The `break` statement stops the code from continuing to run into the next cases.
- If no cases match, the `default` case runs (optional but recommended).

Real-World Example: Day of the Week

Imagine you want to display a message based on the current day:

Full runnable code:

```
let day = "Tuesday";

switch(day) {
  case "Monday":
    console.log("Start of the work week.");
    break;
  case "Tuesday":
    console.log("Second day of the work week.");
    break;
  case "Wednesday":
    console.log("Midweek day.");
    break;
  case "Thursday":
  case "Friday":
    console.log("Almost weekend!");
    break;
  case "Saturday":
  case "Sunday":
    console.log("Weekend!");
    break;
  default:
    console.log("Invalid day.");
}
```

Important Notes

- **`break`** prevents "fall-through," which means without it, the program continues executing all the following cases.
- **`default`** runs when none of the cases match and helps handle unexpected input.

Summary

`switch` is a powerful tool to replace complex `if-else` chains when you're comparing a single variable to many values. Using `break` correctly and including a `default` ensures your code behaves as expected and is easy to read.

5.3 Ternary Operator

The ternary operator is a compact way to write simple `if-else` statements in JavaScript. It's called "ternary" because it takes three parts: a condition, a result if the condition is true, and a result if the condition is false.

Syntax:

```
condition ? expr1 : expr2
```

- If `condition` is true, the operator returns `expr1`.
- If `condition` is false, it returns `expr2`.

Example: Greeting Based on Time

Full runnable code:

```
let hour = 14;
let greeting = hour < 12 ? "Good morning!" : "Good afternoon!";
console.log(greeting);  // Outputs: Good afternoon!
```

This is equivalent to:

Full runnable code:

```
let hour = 14;
let greeting;

if (hour < 12) {
  greeting = "Good morning!";
} else {
  greeting = "Good afternoon!";
}

console.log(greeting);
```

Example: Access Based on Age

Full runnable code:

```
let age = 20;
let access = age >= 18 ? "Access granted" : "Access denied";
console.log(access);  // Outputs: Access granted
```

Why Use the Ternary Operator?

- It makes your code shorter and often easier to read for simple decisions.
- Best used when the result is assigned to a variable or returned directly.
- Avoid overusing it for complex conditions to keep your code clear.

Summary

The ternary operator provides a neat and concise way to express simple conditional logic, making your JavaScript code cleaner and more efficient.

Chapter 6.

Loops and Iteration

1. `for`, `while`, and `do...while` Loops
2. `break` and `continue`
3. Looping Patterns (Countdowns, Accumulators)

6 Loops and Iteration

6.1 `for`, `while`, and `do...while` Loops

Loops are fundamental in JavaScript for running the same block of code multiple times. The three main types — `for`, `while`, and `do...while` — each have their own syntax and best use cases.

`for` Loop

The `for` loop is great when you know **how many times** you want to run the loop, such as counting up or down.

Syntax:

```
for (initialization; condition; update) {
  // code to run
}
```

Example: Counting up from 1 to 5

Full runnable code:

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
// Output: 1 2 3 4 5
```

Example: Counting down from 5 to 1

Full runnable code:

```
for (let i = 5; i >= 1; i--) {
  console.log(i);
}
// Output: 5 4 3 2 1
```

`while` Loop

Use a `while` loop when you want to run code **as long as a condition remains true**, but you might not know in advance how many times it will run.

Syntax:

```
while (condition) {
  // code to run
}
```

Example: Counting up until a condition changes

Full runnable code:

```
let count = 1;
while (count <= 5) {
  console.log(count);
  count++;
}
// Output: 1 2 3 4 5
```

`do...while` Loop

The `do...while` loop guarantees the code inside runs **at least once**, then continues as long as the condition is true.

Syntax:

```
do {
  // code to run
} while (condition);
```

Example: Run code once, then repeat while condition holds

Full runnable code:

```
let count = 1;
do {
  console.log(count);
  count++;
} while (count <= 5);
// Output: 1 2 3 4 5
```

6.1.1 When to Use Each Loop

- Use **`for`** loops when you know the exact number of iterations.
- Use **`while`** loops when the number of iterations depends on a dynamic condition.
- Use **`do...while`** loops when you need to run the code block at least once before checking the condition.

Understanding these loops lets you handle many repetitive tasks efficiently, from counting and iterating over data to waiting for certain conditions in your program.

6.2 `break` and `continue`

When working with loops, sometimes you want to control the flow more precisely—either stop the loop early or skip certain iterations. JavaScript provides two keywords to help with this: `break` and `continue`.

`break` — Exit the Loop Early

The `break` statement immediately stops the entire loop and exits it, no matter where it is in the loop cycle.

Example: Exit loop when a specific number is found

Full runnable code:

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    console.log("Number 5 found, stopping loop.");
    break;  // Exit loop here
  }
  console.log(i);
}

// Output:
// 1
// 2
// 3
// 4
// Number 5 found, stopping loop.
```

In this example, the loop stops as soon as it reaches number 5.

`continue` — Skip the Current Iteration

The `continue` statement skips the rest of the code in the current loop iteration and immediately moves to the next one.

Example: Skip even numbers in a range

Full runnable code:

```
for (let i = 1; i <= 10; i++) {
  if (i % 2 === 0) {
    continue;  // Skip even numbers
  }
  console.log(i);
}

// Output: 1 3 5 7 9
```

Here, whenever the number is even, `continue` skips the `console.log(i)` statement, so only odd numbers are printed.

Summary

- Use `break` to **stop** the loop completely when a condition is met.
- Use `continue` to **skip** certain iterations but keep the loop running.

These control statements give you flexible ways to manage loops, making your code more efficient and easier to read.

6.3 Looping Patterns (Countdowns, Accumulators)

Loops are powerful tools that let you repeat tasks efficiently. Here are some common patterns you'll encounter when working with loops, along with clear examples.

Accumulator Pattern: Summing Numbers

An **accumulator** keeps a running total (or result) that updates each time the loop runs. This is useful for adding up numbers or combining data.

Full runnable code:

```
let sum = 0; // Initialize accumulator

for (let i = 1; i <= 5; i++) {
  sum += i; // Add current number to sum
}

console.log("Sum of numbers 1 to 5 is:", sum);
// Output: Sum of numbers 1 to 5 is: 15
```

Countdown Timer

Loops can also count down from a starting number to zero, useful for timers or countdown displays.

Full runnable code:

```
let countdown = 5;

while (countdown > 0) {
  console.log("Countdown:", countdown);
  countdown--; // Decrement the counter
}

console.log("Liftoff!");
// Output:
// Countdown: 5
// Countdown: 4
// Countdown: 3
// Countdown: 2
// Countdown: 1
// Liftoff!
```

Building a New Array

You can use loops to create or transform arrays by adding items one by one.

Full runnable code:

```
let numbers = [1, 2, 3, 4, 5];
let squares = []; // Empty array to store results

for (let i = 0; i < numbers.length; i++) {
```

```
  squares.push(numbers[i] * numbers[i]); // Square each number and add to new array
}

console.log("Squares:", squares);
// Output: Squares: [1, 4, 9, 16, 25]
```

6.3.1 Summary

- **Accumulators** keep track of totals or combined values across iterations.
- **Countdowns** decrease a value until a condition is met.
- Loops can **build or transform arrays** by adding values dynamically.

These patterns are essential building blocks for solving many programming problems and will help you write efficient, readable code.

Chapter 7.

Arrays and Array Methods

1. Creating and Accessing Arrays
2. Common Array Methods (`push`, `pop`, `shift`, `unshift`, `slice`, `splice`)
3. Iterating Arrays with Loops
4. Advanced Array Methods (`map`, `filter`, `reduce`, `forEach`, `find`)

7 Arrays and Array Methods

7.1 Creating and Accessing Arrays

Arrays are one of the most important data structures in JavaScript. They allow you to store multiple values in a single variable, making it easy to manage collections of data such as lists of numbers, strings, objects, or even other arrays.

What Is an Array?

An array is an **ordered collection** of items where each item can be accessed by its **index**, which starts at 0. Arrays are especially useful when you need to work with lists of related values — for example, a list of student names, a series of temperatures, or a shopping cart's contents.

Creating Arrays

There are two common ways to create arrays in JavaScript:

1. Using Array Literals (Recommended)

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

This is the most concise and preferred way to create an array. The array `fruits` contains three string elements.

2. Using the Array Constructor

```
let numbers = new Array(10, 20, 30);
```

This creates an array `numbers` with the elements `10`, `20`, and `30`.

You can also create an empty array:

```
let emptyArray = [];
```

Or use the constructor with a specified length:

```
let scores = new Array(5); // creates an array with 5 empty slots
```

> WARNING Be careful: `new Array(5)` creates an array with *length 5* but doesn't assign values to any of the elements.

Accessing Array Elements

You access elements in an array by referring to their index (starting from 0):

Full runnable code:

```
let colors = ['red', 'green', 'blue'];
console.log(colors[0]); // Output: 'red'
console.log(colors[2]); // Output: 'blue'
```

To modify a specific element:

Full runnable code:

```
let colors = ['red', 'green', 'blue'];
colors[1] = 'yellow';
console.log(colors); // ['red', 'yellow', 'blue']
```

Accessing Out-of-Range Indexes

If you try to access an index that doesn't exist in the array, JavaScript will return `undefined`:

```
let pets = ['dog', 'cat'];
console.log(pets[5]); // Output: undefined
```

Assigning a value to an out-of-range index expands the array and fills missing spots with `undefined`:

```
pets[4] = 'hamster';
console.log(pets); // ['dog', 'cat', undefined, undefined, 'hamster']
```

Summary

Arrays are essential for working with ordered data in JavaScript. You can create them using literals or constructors and access items using numeric indexes. Being aware of how JavaScript handles array lengths and out-of-range indexes will help you write more reliable and efficient code.

7.2 Common Array Methods (`push`, `pop`, `shift`, `unshift`, `slice`, `splice`)

JavaScript provides several built-in methods that make working with arrays easier and more powerful. In this section, we'll explore six essential array methods that help you add, remove, copy, and modify elements.

`push()`

Purpose: Adds one or more elements to the **end** of an array.

Syntax:

```
array.push(element1, element2, ...);
```

Example:

Full runnable code:

```
let fruits = ['apple', 'banana'];
fruits.push('cherry');
console.log(fruits); // ['apple', 'banana', 'cherry']
```

Use Case: Adding new items to the end of a list.

`pop()`

Purpose: Removes the **last** element from an array and returns it.

Syntax:

```
let removedItem = array.pop();
```

Example:

Full runnable code:

```
let fruits = ['apple', 'banana', 'cherry'];
let lastFruit = fruits.pop();
console.log(lastFruit); // 'cherry'
console.log(fruits);    // ['apple', 'banana']
```

Use Case: Removing the most recent item from the end.

`shift()`

Purpose: Removes the **first** element from an array and returns it.

Syntax:

```
let removedItem = array.shift();
```

Example:

Full runnable code:

```
let colors = ['red', 'green', 'blue'];
let firstColor = colors.shift();
console.log(firstColor); // 'red'
console.log(colors);     // ['green', 'blue']
```

Use Case: Processing a queue (FIFO — first in, first out).

`unshift()`

Purpose: Adds one or more elements to the **beginning** of an array.

Syntax:

```
array.unshift(element1, element2, ...);
```

Example:

Full runnable code:

```
let colors = ['green', 'blue'];
colors.unshift('red');
console.log(colors); // ['red', 'green', 'blue']
```

Use Case: Prepending items to a list.

`slice()`

Purpose: Returns a **shallow copy** of a portion of an array without modifying the original array.

Syntax:

```
array.slice(startIndex, endIndex);
```

- `startIndex`: Index to start (inclusive)
- `endIndex`: Index to stop (exclusive)

Example:

Full runnable code:

```
let numbers = [10, 20, 30, 40, 50];
let subArray = numbers.slice(1, 4);
console.log(subArray); // [20, 30, 40]
console.log(numbers);  // [10, 20, 30, 40, 50] (unchanged)
```

Use Case: Copying a portion of an array without affecting the original.

`splice()`

Purpose: Changes an array by **adding, removing, or replacing** elements in place.

Syntax:

```
array.splice(startIndex, deleteCount, item1, item2, ...);
```

- `startIndex`: Index to begin changes
- `deleteCount`: Number of elements to remove
- `item1, item2, ...`: Elements to add (optional)

Example 1 – Removing Elements:

Full runnable code:

```
let items = ['a', 'b', 'c', 'd'];
items.splice(1, 2);
console.log(items); // ['a', 'd']
```

Example 2 – Adding Elements:

Full runnable code:

```
let items = ['a', 'd'];
items.splice(1, 0, 'b', 'c');
console.log(items); // ['a', 'b', 'c', 'd']
```

Example 3 – Replacing Elements:

Full runnable code:

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

Use Case: Directly modifying part of an array, such as editing a to-do list.

Summary Table

Method	Action	Modifies Original?	Common Use
`push()`	Add to end	YES Yes	Stack: Add
`pop()`	Remove from end	YES Yes	Stack: Remove
`shift()`	Remove from start	YES Yes	Queue: Dequeue
`unshift()`	Add to start	YES Yes	Queue: Enqueue
`slice()`	Copy portion (no change)	NO No	Cloning subarrays
`splice()`	Add/Remove/Replace in-place	YES Yes	Editing arrays

Understanding these methods gives you powerful tools to manipulate arrays efficiently in a wide variety of practical programming tasks.

7.3 Iterating Arrays with Loops

Iterating over arrays—going through each item one by one—is a common task in programming. JavaScript offers several types of loops to help you work with array elements effectively. In this section, we'll look at the most commonly used loop types: `for`, `while`, and `for...of`.

7.3.1 The Classic `for` Loop

Purpose: Gives full control over the iteration process using an index.

Syntax:

```
for (let i = 0; i < array.length; i++) {
  // Use array[i]
}
```

Example – Summing Numbers:

Full runnable code:

```
let numbers = [10, 20, 30];
let sum = 0;

for (let i = 0; i < numbers.length; i++) {
```

```
  sum += numbers[i];
}
console.log(sum); // 60
```

Common Mistake: Off-by-one errors like using `i <= array.length` instead of `i < array.length`, which will try to access an out-of-range index.

7.3.2 The `while` Loop

Purpose: Repeats a block of code while a condition is true. Useful when the number of iterations isn't fixed or depends on dynamic conditions.

Syntax:

```
let i = 0;
while (i < array.length) {
  // Use array[i]
  i++;
}
```

Example – Printing Elements:

Full runnable code:

```
let fruits = ['apple', 'banana', 'cherry'];
let i = 0;

while (i < fruits.length) {
  console.log(fruits[i]);
  i++;
}
// Output:
// apple
// banana
// cherry
```

Tip: Always make sure your loop has an end condition. Forgetting to increment `i` can cause an infinite loop.

7.3.3 The `for...of` Loop

Purpose: A modern, concise way to loop through array values **without using indexes**. Improves readability and avoids index-related bugs.

Syntax:

```
for (let element of array) {
  // Use element
```

```
}
```

Example – Capitalizing Words:

Full runnable code:

```
let animals = ['cat', 'dog', 'fox'];

for (let animal of animals) {
  console.log(animal.toUpperCase());
}
// Output:
// CAT
// DOG
// FOX
```

Use Case: When you just need the values, not the index.

7.3.4 Comparing Loop Styles

Loop Type	Best When You Need…	Has Index?	Readability
`for`	Full control over start/stop/index	YES Yes	Moderate
`while`	Dynamic stopping conditions	YES Yes	Low–Moderate
`for...of`	Simple value-based iteration	NO No	YES High

7.3.5 Beginner Tips

- **Avoid `i <= array.length`** — it will go one step too far.
- **Use `for...of`** when you don't need the index—it's cleaner and safer.
- **Keep loops focused** — avoid doing too much inside a loop block.
- **Use `const` inside `for...of`** if the value doesn't change:

```
for (const name of names) {
  console.log(name);
}
```

7.3.6 Summary

Looping through arrays is essential for tasks like summing values, transforming data, and displaying results. Understanding how and when to use different loop types will help you write cleaner, more efficient JavaScript. Start with `for...of` for simplicity, and use classic loops when more control is needed.

7.4 Advanced Array Methods (`map`, `filter`, `reduce`, `forEach`, `find`)

JavaScript arrays come with powerful **higher-order methods** that allow you to write cleaner, more expressive code. These methods take **callback functions** as arguments and help you **transform**, **filter**, **summarize**, and **search** data efficiently.

Let's explore each one with practical, real-world examples.

7.4.1 `map()`: Transforming Data

Purpose: Creates a **new array** by applying a function to **each element** of the original array.

Syntax:

```
array.map((element, index, array) => {
  // return transformed value
});
```

Example – Convert prices from USD to EUR:

Full runnable code:

```
let usdPrices = [10, 20, 30];
let eurPrices = usdPrices.map(price => price * 0.9);
console.log(eurPrices); // [9, 18, 27]
```

Use Case: Applying the same transformation to each item (e.g., converting units, formatting text).

7.4.2 `filter()`: Selecting Subsets

Purpose: Creates a **new array** with only the elements that **pass a test** (i.e., return `true` from the callback).

Syntax:

```
array.filter((element, index, array) => {
  return condition;
});
```

Example – Get users over age 18:

Full runnable code:

```
let users = [
  { name: 'Alice', age: 17 },
  { name: 'Bob', age: 22 },
  { name: 'Carol', age: 19 }
];

let adults = users.filter(user => user.age >= 18);
//console.table(adults);

// Pretty-print with indentation
console.log(JSON.stringify(adults, null, 2));

// [
//   { name: 'Bob', age: 22 },
//   { name: 'Carol', age: 19 }
// ]
```

Use Case: Keeping only the items that match a condition (e.g., active users, valid inputs).

7.4.3 `reduce()`: Aggregating Results

Purpose: Reduces an array to a **single value** by applying a function cumulatively.

Syntax:

```
array.reduce((accumulator, current, index, array) => {
  return updatedAccumulator;
}, initialValue);
```

Example – Calculate total cart price:

Full runnable code:

```
let cart = [29.99, 9.99, 4.99];
let total = cart.reduce((sum, price) => sum + price, 0);
console.log(total); // 44.97
```

Use Case: Summing values, counting occurrences, or building a new object from array data.

7.4.4 `forEach()`: Executing Side Effects

Purpose: Executes a function **on each element** in the array. Unlike `map`, it **does not return a new array**.

Syntax:

```
array.forEach((element, index, array) => {
  // side effect (e.g., logging)
});
```

Example – Log each order ID:

Full runnable code:

```
let orders = [101, 102, 103];
orders.forEach(id => console.log(`Order #${id}`));
// Output:
// Order #101
// Order #102
// Order #103
```

Use Case: Performing operations like logging, DOM updates, or API calls.

> WARNING **Tip:** Don't use `forEach()` when you need to transform data—use `map()` instead.

7.4.5 `find()`: Locating an Element

Purpose: Returns the **first element** that satisfies a condition. If none is found, returns `undefined`.

Syntax:

```
array.find((element, index, array) => {
  return condition;
});
```

Example – Find the first overdue task:

Full runnable code:

```
let tasks = [
  { title: 'Pay bills', done: true },
  { title: 'Clean room', done: false },
  { title: 'Buy groceries', done: false }
];

let firstUndone = tasks.find(task => !task.done);
console.log(JSON.stringify(firstUndone, null, 2));
// { title: 'Clean room', done: false }
```

Use Case: Searching for a specific match (e.g., user by ID, product by name).

7.4.6 Summary Table

Method	Purpose	Returns	Mutates?	Best For
`map()`	Transform elements	New array	NO	Format/change array values
`filter()`	Select elements	New array	NO	Get a subset of items
`reduce()`	Aggregate to single value	Single value	NO	Sum, count, or combine items
`forEach()`	Perform side effects	`undefined`	NO	Logging, DOM manipulation
`find()`	Find first match	Single item/`undefined`	NO	Locate an object or value

Mastering these advanced methods lets you write elegant, efficient code for complex data tasks. With practice, you'll find they often replace traditional loops entirely in modern JavaScript development.

Chapter 8.

Objects and Object Literals

1. Creating Objects
2. Accessing and Updating Properties
3. Nesting Objects
4. Iterating Over Objects

8 Objects and Object Literals

8.1 Creating Objects

In JavaScript, **objects** are one of the most powerful and flexible data structures. They allow you to group related data and behavior together using a **key-value** format.

Where arrays are ideal for **ordered lists** of items (like a list of names or scores), objects are best used when you want to represent **structured data** — like a person, a car, or a product — where each piece of data is identified by a descriptive name (a key).

8.1.1 What Is an Object?

An object is a **collection of properties**, where each property has a **key (or property name)** and a **value**. These keys are always strings (or symbols), and values can be **any type of data**, including strings, numbers, arrays, other objects, or functions.

8.1.2 Creating Objects with Object Literals (Preferred)

The most common and readable way to create an object is by using **object literal syntax** — simply wrap the key-value pairs inside curly braces `{}`.

Example – Creating a Person Object:

```
let person = {
  name: 'Alice',
  age: 30,
  isStudent: false
};
```

Here:

- `name`, `age`, and `isStudent` are **keys (or property names)**
- `'Alice'`, `30`, and `false` are the corresponding **values**

You can also create an empty object and add properties later:

```
let book = {};
book.title = 'JavaScript Basics';
book.pages = 250;
```

8.1.3 Creating Objects with `new Object()` Constructor

JavaScript also allows you to create objects using the `new Object()` constructor. This approach is more verbose and rarely used today in favor of object literals.

Example – Using Constructor:

```
let car = new Object();
car.make = 'Toyota';
car.model = 'Camry';
car.year = 2022;
```

This produces the same result as:

```
let car = {
  make: 'Toyota',
  model: 'Camry',
  year: 2022
};
```

8.1.4 When to Use Objects vs. Arrays

Use Case	Choose…	Example
Ordered list of values	**Array**	`['apple', 'banana', 'cherry']`
Named pieces of related data	**Object**	`{ name: 'Alice', age: 30 }`
Need to access items by position	**Array**	`colors[0]`
Need to access items by name	**Object**	`person.name`

Objects shine when you want to **group and label data**, especially when that data describes attributes or characteristics of a thing.

8.1.5 Summary

- **Objects** store data as **key-value pairs**.
- The **object literal syntax** (`{}`) is the most common and readable way to create objects.
- The **`new Object()`** method is less common and usually avoided in modern code.
- Use **objects** when your data is structured and best represented by named properties.

As you continue building JavaScript programs, you'll use objects frequently to represent real-world entities and store flexible, descriptive data.

8.2 Accessing and Updating Properties

Once you've created an object, you'll often need to **read**, **update**, **add**, or even **delete** its properties. JavaScript provides two main ways to access and modify object properties: **dot notation** and **bracket notation**.

8.2.1 Accessing Properties

Dot Notation (Recommended for Known Property Names)

Syntax:

```
object.propertyName
```

Example:

Full runnable code:

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

console.log(person.name); // 'Alice'
console.log(person.age);  // 30
```

Dot notation is simple and readable, but it only works when:

- The property name is a valid identifier (no spaces or special characters)
- You know the property name ahead of time

Bracket Notation (For Dynamic or Unusual Keys)

Syntax:

```
object['propertyName']
```

Example with space in key:

Full runnable code:

```
let book = {
  title: 'JavaScript Guide',
  'page count': 300
};

console.log(book['page count']); // 300
```

Example with a dynamic key:

Full runnable code:

```
let key = 'name';
let person = { name: 'Bob' };

console.log(person[key]); // 'Bob'
```

Use bracket notation when the property name is stored in a variable or contains spaces/special characters.

8.2.2 Updating Properties

You can change the value of an existing property using either dot or bracket notation.

Example – Updating a Property:

Full runnable code:

```
let car = {
  make: 'Toyota',
  year: 2020
};

car.year = 2023;
car['make'] = 'Honda';

console.log(JSON.stringify(car,null,2)); // { make: 'Honda', year: 2023 }
```

8.2.3 Adding New Properties

If the property doesn't exist, assigning a value will **create** it:

Full runnable code:

```
let user = { username: 'jsLearner' };

user.age = 25;
user['email'] = 'user@example.com';

console.log(JSON.stringify(user,null,2));
// { username: 'jsLearner', age: 25, email: 'user@example.com' }
```

8.2.4 Deleting Properties

You can remove a property using the `delete` operator:

Full runnable code:

```
let profile = {
  name: 'Charlie',
  location: 'Canada'
};

delete profile.location;

console.log(JSON.stringify(profile,null,2)); // { name: 'Charlie' }
```

> WARNING Note: `delete` only removes the property from the object; it doesn't affect the prototype.

8.2.5 Summary

Task	Preferred Notation	Example
Read known property	Dot notation	`obj.name`
Read dynamic/odd key	Bracket notation	`obj['full name']` or `obj[key]`
Update property	Dot or bracket	`obj.age = 30`
Add new property	Dot or bracket	`obj.city = 'Paris'`
Delete property	`delete` operator	`delete obj.city`

Understanding how to access and update properties is fundamental when working with objects. These skills allow you to build and manipulate data structures that represent real-world entities and change over time.

8.3 Nesting Objects

In JavaScript, objects can contain other **objects** or **arrays** as property values. This is called **nesting**, and it's a powerful way to model **complex, structured data** — like user profiles, product catalogs, or app settings.

Nested objects help you organize related data in a way that mirrors real-world relationships.

8.3.1 Example: A User Profile Object

Let's say we want to model a user with personal info, an address, and preferences. We can nest objects to represent each section:

```
let user = {
  name: 'Emily',
  age: 28,
  address: {
    street: '123 Main St',
    city: 'New York',
    zip: '10001'
  },
  preferences: {
    theme: 'dark',
    notifications: true
  }
};
```

Here:

- `address` and `preferences` are objects **inside** the main `user` object.
- Each nested object has its own set of key-value pairs.

8.3.2 Accessing Nested Properties

You can use **dot notation** or **bracket notation** to reach properties inside nested objects.

Examples:

```
console.log(user.address.city);          // 'New York'
console.log(user.preferences.theme);     // 'dark'
```

If a key has spaces or is stored in a variable, use bracket notation:

```
console.log(user['address']['zip']);     // '10001'
```

8.3.3 Updating Nested Properties

You can assign new values to nested properties just like top-level ones:

```
user.address.city = 'Los Angeles';
user.preferences.theme = 'light';

console.log(user.address.city);          // 'Los Angeles'
console.log(user.preferences.theme);     // 'light'
```

8.3.4 Adding Properties to Nested Objects

You can also add new properties at any level:

```
user.address.country = 'USA';
user.preferences.language = 'English';

console.log(user.address.country);        // 'USA'
console.log(user.preferences.language);   // 'English'
```

8.3.5 Nested Arrays in Objects

It's also common to store arrays inside objects:

Full runnable code:

```
let product = {
  name: 'Laptop',
  price: 999,
  features: ['touchscreen', 'backlit keyboard', 'USB-C']
};

console.log(product.features[1]); // 'backlit keyboard'
```

Or even objects **inside arrays**:

Full runnable code:

```
let order = {
  id: 1234,
  items: [
    { name: 'Phone', quantity: 1 },
    { name: 'Charger', quantity: 2 }
  ]
};

console.log(order.items[1].name); // 'Charger'
```

8.3.6 Summary

- Objects can contain **other objects or arrays** to model real-world structures.
- Use **dot or bracket notation** to access and update deeply nested properties.
- Nesting improves organization and scalability for complex data.

By mastering nested structures, you'll be able to represent rich data models and work confidently with modern APIs, user data, or configuration objects in your applications.

8.4 Iterating Over Objects

In JavaScript, iterating over an object means going through its **keys and values**—a common task when you need to inspect, transform, or manipulate data dynamically. Unlike arrays, objects don't have a built-in index-based loop, but JavaScript provides several ways to loop through object properties.

In this section, we'll explore:

- `for...in` loops
- `Object.keys()`
- `Object.values()`
- `Object.entries()`

8.4.1 `for...in`: Looping Over Keys

The `for...in` loop lets you iterate over all **enumerable properties** of an object.

Syntax:

```
for (let key in object) {
  // Use key and object[key]
}
```

Example – Print all key-value pairs:

Full runnable code:

```
let person = {
  name: 'Alice',
  age: 30,
  country: 'Canada'
};

for (let key in person) {
  console.log(key + ': ' + person[key]);
}
// Output:
// name: Alice
// age: 30
// country: Canada
```

> WARNING Note: `for...in` loops also iterate over inherited properties, so it's good practice to check ownership with `hasOwnProperty()` if needed.

8.4.2 `Object.keys()`: Getting All Keys

`Object.keys()` returns an **array of property names** (keys) from the object.

Example:

Full runnable code:

```
let car = {
  make: 'Honda',
  model: 'Civic',
  year: 2022
};

let keys = Object.keys(car);
console.log(keys); // ['make', 'model', 'year']
```

You can combine this with a `for` or `forEach()` loop:

```
Object.keys(car).forEach(key => {
  console.log(`${key}: ${car[key]}`);
});
```

8.4.3 `Object.values()`: Getting All Values

`Object.values()` returns an array of the object's values only.

Example:

Full runnable code:

```
let scores = {
  math: 90,
  english: 85,
  science: 95
};

let values = Object.values(scores);
console.log(values); // [90, 85, 95]
```

You can sum the values:

```
let total = Object.values(scores).reduce((sum, score) => sum + score, 0);
console.log(total); // 270
```

8.4.4 `Object.entries()`: Getting Key-Value Pairs

`Object.entries()` returns an array of `[key, value]` pairs, making it ideal for looping with `for...of`.

Example – Loop with `for...of`:

Full runnable code:

```
let user = {
  username: 'coder123',
  level: 'beginner',
  active: true
};

for (let [key, value] of Object.entries(user)) {
  console.log(`${key} => ${value}`);
}
// Output:
// username => coder123
// level => beginner
// active => true
```

8.4.5 Use Cases

Task	Recommended Method
List keys	`Object.keys()` or `for...in`
List values	`Object.values()`
Loop through key-value pairs	`Object.entries()`
Copy or transform properties	Combine `Object.keys()` with mapping logic

8.4.6 Summary

Method	Returns	Ideal For
`for...in`	Each key (one at a time)	Simple loops (use with caution)
`Object.keys(obj)`	Array of keys	Accessing all property names
`Object.values(obj)`	Array of values	Summing or listing values
`Object.entries(obj)`	Array of `[key, value]` pairs	Full key-value iteration

Mastering these object iteration techniques enables you to dynamically inspect, transform, or display structured data—an essential skill when working with user profiles, configuration settings, API responses, or any real-world data structure.

Chapter 9.

Functions and Scope

1. Declaring Functions (`function`, arrow functions)
2. Parameters and Return Values
3. Function Expressions
4. Variable Scope and Hoisting
5. Closures (Introductory)

9 Functions and Scope

9.1 Declaring Functions (`function`, arrow functions)

Functions are one of the most fundamental building blocks in JavaScript. They allow you to **group reusable code** into named blocks that can be called whenever needed. JavaScript supports multiple ways to declare functions, with two of the most common being:

- **Function declarations** (traditional)
- **Arrow functions** (introduced in ES6)

In this section, you'll learn how to use both styles, understand their differences, and see when each is most appropriate.

9.1.1 Traditional Function Declarations

The traditional way to declare a function uses the `function` keyword.

Syntax:

```
function functionName(parameters) {
  // code to execute
  return result;
}
```

Example – Add two numbers:

Full runnable code:

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

console.log(add(3, 4)); // 7
```

Example – Greet a user:

Full runnable code:

```
function greet(name) {
  return `Hello, ${name}!`;
}

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

9.1.2 Arrow Functions

Arrow functions offer a **shorter syntax** and are commonly used in modern JavaScript, especially for small or anonymous functions.

Syntax:

```
const functionName = (parameters) => {
  // code to execute
  return result;
};
```

If the function has only **one expression**, you can omit the braces and `return` keyword:

```
const add = (a, b) => a + b;
```

Example – Greet a user (arrow version):

Full runnable code:

```
const greet = name => `Hello, ${name}!`;

console.log(greet('Bob')); // Hello, Bob!
```

9.1.3 Comparing Function Declarations vs. Arrow Functions

Feature	Function Declaration	Arrow Function
Syntax	Verbose	Concise
`this` binding	Dynamic (`this` depends on caller)	Lexical (`this` is inherited from scope)
Hoisting	Yes (can be used before defined)	No (must be declared first)
Best for	Methods, complex logic	Callbacks, one-liners

9.1.4 Important: `this` Behavior

One of the biggest differences between the two styles is how they handle the `this` keyword.

Function Declaration:

Full runnable code:

```
const user = {
  name: 'Alice',
  greet: function () {
    console.log(`Hi, I'm ${this.name}`);
  }
```

```
};

user.greet(); // Hi, I'm Alice
```

Arrow Function (wrong in this case):

Full runnable code:

```
const user = {
  name: 'Bob',
  greet: () => {
    console.log(`Hi, I'm ${this.name}`);
  }
};

user.greet(); // Hi, I'm undefined
```

> Arrow functions do **not** have their own `this`. They inherit it from the surrounding context, which can lead to unexpected results when used in object methods.

9.1.5 Summary

- Use **function declarations** when you need hoisting or when defining methods inside objects.
- Use **arrow functions** for **shorter syntax**, especially in callbacks, array methods, or simple functions.
- Be careful with `this` — prefer function declarations for methods that rely on object context.

By mastering both styles, you'll be able to write clean, modern JavaScript code that's both expressive and efficient.

9.2 Parameters and Return Values

Functions in JavaScript behave much like **machines**: you give them **input** (parameters), they perform a task, and they may produce **output** (a return value). Understanding how to pass data into functions and get results back is essential to writing flexible, reusable code.

9.2.1 What Are Parameters?

Parameters are like variables listed in a function's definition. They represent values that will be passed **into the function** when it's called.

Arguments are the actual values you pass when calling the function.

9.2.2 Functions with No Parameters

Sometimes, a function doesn't need any input — it simply performs an action.

Example – Say Hello:

Full runnable code:

```
function sayHello() {
  console.log('Hello!');
}

sayHello(); // Output: Hello!
```

9.2.3 Functions with One Parameter

Example – Greet a user by name:

Full runnable code:

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

greet('Alice'); // Output: Hello, Alice!
```

Here, `name` is a parameter, and `'Alice'` is the argument passed to it.

9.2.4 Functions with Multiple Parameters

You can define as many parameters as you need, separated by commas.

Example – Add two numbers:

Full runnable code:

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

let result = add(5, 3);
console.log(result); // 8
```

9.2.5 Default Parameter Values

If a caller doesn't provide an argument, you can define a **default value** to use instead.

Example – Greet with a default name:

Full runnable code:

```
function greet(name = 'Guest') {
  console.log(`Welcome, ${name}!`);
}

greet();          // Welcome, Guest!
greet('Charlie'); // Welcome, Charlie!
```

This makes your functions more **robust** and **error-tolerant**.

9.2.6 Returning Values

A function can return a value using the `return` keyword. This lets you **store or use the result** of a computation later in your code.

Example – Calculate area of a rectangle:

Full runnable code:

```
function calculateArea(width, height) {
  return width * height;
}

let area = calculateArea(5, 4);
console.log(area); // 20
```

Without `return`, the function would perform the calculation but not give the result back to the code that called it.

9.2.7 Real-World Analogy

Think of a function like a **coffee machine**:

- **Parameters** = the inputs (e.g. water, coffee beans, sugar)
- **Function body** = the brewing process
- **Return value** = the finished cup of coffee

If you press the button without choosing sugar, maybe it adds none (default value). When the brewing is done, you get coffee (the return value), which you can drink (use in your program).

9.2.8 Summary

Concept	Example	Purpose
No parameters	`function sayHi() {}`	Executes fixed code
One parameter	`function greet(name) {}`	Accepts a single value
Multiple parameters	`function add(a, b) {}`	Accepts two or more values
Default values	`function greet(name = 'Guest') {}`	Fallback when no argument passed
Return a value	`return a + b;`	Outputs a result from the function

Knowing how to pass data into functions and return results lets you build **custom, reusable tools** for your code—just like mini-programs inside your program.

9.3 Function Expressions

In JavaScript, functions are **first-class citizens**, meaning they can be treated like any other value — stored in variables, passed as arguments, or returned from other functions. One way to create functions that leverages this flexibility is through **function expressions**.

9.3.1 What Is a Function Expression?

A **function expression** defines a function inside an expression, typically assigning it to a variable.

Syntax:

```
const sayHello = function() {
  console.log('Hello!');
};
```

Unlike function declarations (`function sayHello() {}`), function expressions are **not hoisted**, which means you cannot call them before they are defined.

9.3.2 Example: Storing Functions in Variables

You can store a function in a variable just like a number or string.

Full runnable code:

```
const multiply = function(a, b) {
  return a * b;
};

console.log(multiply(4, 5)); // 20
```

9.3.3 Passing Functions as Arguments (Callbacks)

Because functions can be stored in variables, they can also be **passed as arguments** to other functions. This is useful for things like event handling or array processing.

Example – Using a callback with `setTimeout`:

Full runnable code:

```
setTimeout(function() {
  console.log('This message appears after 2 seconds');
}, 2000);
```

Here, the anonymous function (a function without a name) is passed directly as an argument to `setTimeout`.

9.3.4 Anonymous Functions

Functions without a name are called **anonymous functions**. They are often used in function expressions or as callbacks:

Full runnable code:

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

greet('Alice'); // Hello, Alice!
```

You can also write anonymous arrow functions:

Full runnable code:

```
setTimeout(() => console.log('Done!'), 1000);
```

9.3.5 Function Declarations vs. Function Expressions

Feature	Function Declaration	Function Expression
Syntax	`function foo() {}`	`const foo = function() {}`
Hoisting	Hoisted (can call before defined)	Not hoisted (must define first)
Naming	Usually named	Can be anonymous or named
Usage	General purpose	When assigning or passing functions

9.3.6 Summary

- **Function expressions** create functions inside expressions, often assigned to variables.
- They **are not hoisted**, so define them before use.
- They can be **anonymous**, making them handy as inline callbacks.
- Function expressions enable powerful patterns like passing functions as arguments or returning functions from other functions.

Understanding function expressions unlocks much of JavaScript's flexibility and lets you write more dynamic and functional code.

9.4 Variable Scope and Hoisting

Understanding **variable scope** and **hoisting** is essential for writing reliable JavaScript code. These concepts determine **where variables are accessible** and **how they behave during code execution**.

9.4.1 What Is Variable Scope?

Scope defines the **region of your code where a variable is available**. In JavaScript, there are three main types of scope:

- **Global scope**
- **Function scope**
- **Block scope**

9.4.2 Global Scope

Variables declared outside any function or block have **global scope** and can be accessed anywhere in your code.

Full runnable code:

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

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

showVar();                // Output: I am global
console.log(globalVar);  // Output: I am global
```

9.4.3 Function Scope

Variables declared inside a function are only accessible **within that function**.

Full runnable code:

```
function myFunc() {
  let funcVar = 'I am local to myFunc';
  console.log(funcVar); // Works here
}

myFunc();
console.log(funcVar); // Error: funcVar is not defined
```

Variables declared with `var` and `let` inside functions are both function-scoped, meaning they exist throughout the function.

9.4.4 Block Scope

Block scope means variables are only accessible **inside the nearest pair of `{}`**, such as in `if` statements, loops, or blocks.

- `let` and `const` are **block-scoped**.
- `var` is **not block-scoped** — it's function-scoped.

Example:

Full runnable code:

```
if (true) {
  let blockVar = 'I exist only in this block';
  var varVar = 'I am function-scoped';
}

console.log(blockVar); // Error: blockVar is not defined
console.log(varVar);   // Output: I am function-scoped
```

9.4.5 `var` vs. `let` vs. `const`

Keyword	Scope	Can be Reassigned?	Can be Redeclared in Same Scope?
`var`	Function scope	Yes	Yes
`let`	Block scope	Yes	No
`const`	Block scope	No (constant)	No

Example of `let` and `const`:

```
let count = 1;
count = 2; // Allowed

const max = 10;
max = 20; // Error: Assignment to constant variable
```

9.4.6 What Is Hoisting?

Hoisting is JavaScript's behavior of moving **variable and function declarations** to the top of their scope during compilation, **before code runs**.

This means you can sometimes use variables and functions before they are declared — but the details vary depending on how they were declared.

9.4.7 Hoisting with `var`

`var` declarations are hoisted and **initialized with `undefined`**.

```
console.log(myVar); // Output: undefined
var myVar = 10;
```

Behind the scenes, JavaScript treats it like:

```
var myVar;
console.log(myVar); // undefined
myVar = 10;
```

9.4.8 Hoisting with `let` and `const`

`let` and `const` declarations are hoisted **but not initialized**. Accessing them before declaration causes a **ReferenceError** due to the **Temporal Dead Zone (TDZ)**.

Full runnable code:

```
console.log(myLet); // ReferenceError
let myLet = 5;
```

9.4.9 Hoisting with Functions

- **Function declarations** are fully hoisted and can be called before their definition.

Full runnable code:

```
sayHi(); // Works!

function sayHi() {
  console.log('Hi!');
}
```

- **Function expressions** (including arrow functions) behave like variables declared with `var`, `let`, or `const` depending on how they are assigned.

Full runnable code:

```
sayHello(); // Error or undefined behavior depending on declaration

const sayHello = function() {
  console.log('Hello!');
};
```

9.4.10 Summary

Concept	Description	Example
Global scope	Variables available anywhere	`let x = 5;`
Function scope	Variables available only inside a function	`function() { var y = 10; }`
Block scope	Variables available only inside `{}` blocks (`let`, `const`)	`if (true) { let z = 3; }`
Hoisting	Declarations moved to top; `var` initialized to `undefined`, `let`/`const` not initialized	`console.log(a); var a = 1;`
Temporal Dead Zone (TDZ)	Time before `let` or `const` declaration when accessing them throws an error	`console.log(b); let b = 2;`

Understanding scope and hoisting helps you avoid bugs, write cleaner code, and debug issues related to variable accessibility and initialization.

9.5 Closures (Introductory)

A **closure** is a powerful and important concept in JavaScript. Simply put, a closure is when a function **remembers the environment in which it was created**, even after that environment has finished executing.

This might sound complicated, but closures let you write functions that keep access to variables defined outside themselves — which can be incredibly useful.

9.5.1 What Does a Closure Mean?

When a function is defined inside another function, the inner function **retains access** to the variables of the outer function, even after the outer function has returned. This saved environment is called a **closure**.

9.5.2 Simple Example: A Counter Function

Let's build a function that creates counters — each counter remembers its own count:

Full runnable code:

```
function createCounter() {
  let count = 0;  // This variable is "closed over"

  return function() {
    count++;
    return count;
  };
}

const counter = createCounter();

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

What's happening here?

- `createCounter()` defines a local variable `count` and returns an **anonymous function**.
- That returned function **closes over** `count`, remembering it.
- Even though `createCounter()` finished executing, the inner function still has access to `count`.
- Each time you call `counter()`, it updates and returns the updated `count`.

9.5.3 Another Example: Greeting Generator

Closures also help when you want to generate customized functions:

Full runnable code:

```
function greetGenerator(greeting) {
  return function(name) {
    return `${greeting}, ${name}!`;
  };
}

const sayHello = greetGenerator('Hello');
const sayHi = greetGenerator('Hi');

console.log(sayHello('Alice')); // Hello, Alice!
console.log(sayHi('Bob'));      // Hi, Bob!
```

Here, each generated function remembers the `greeting` passed to the outer function.

9.5.4 Why Are Closures Useful?

- **Data privacy**: Variables inside a closure can't be accessed from outside, protecting data.

- **Function factories**: Create specialized functions dynamically.
- **Maintaining state**: Keep track of data across multiple calls without using global variables.

Closures are everywhere in JavaScript — in event handlers, callbacks, and many libraries — because they let you write more flexible, modular, and powerful code.

9.5.5 Summary

- A **closure** happens when a function remembers its surrounding scope even after the outer function finishes.
- Closures allow inner functions to access variables of outer functions.
- They are useful for **keeping state**, **data privacy**, and **creating customizable functions**.

By understanding closures, you'll unlock deeper JavaScript concepts and write code that's both elegant and effective.

Chapter 10.

Events and User Interaction

1. The DOM Event Model
2. `addEventListener`
3. Handling Click, Keyboard, and Form Events
4. Event Object and Propagation

10 Events and User Interaction

10.1 The DOM Event Model

When you interact with a web page—by clicking buttons, typing in forms, or moving your mouse—these actions are called **events**. The browser detects these events and uses the **DOM Event Model** to respond and update the page accordingly.

10.1.1 Event Sources and Event Listeners

Every interactive element in a web page, such as buttons, links, or input fields, is represented by a **DOM node** (like a branch or leaf on a tree). When something happens to these nodes (a click, keypress, etc.), an **event** is fired.

To react to events, you attach **event listeners** to DOM nodes. An event listener is like a person standing at a branch, waiting to hear if something happens there.

Example:

```
button.addEventListener('click', () => {
  alert('Button clicked!');
});
```

Here, the button is listening for a "click" event and responds by showing an alert.

10.1.2 How Events Travel: Capturing and Bubbling

Events don't just happen on a single element—they **travel through the DOM tree**. Think of the DOM as a family tree:

- The **document** is the root (grandparent)
- Containers are branches (parents)
- Elements are leaves (children)

When an event happens on a leaf, the event goes through three phases:

1. **Capturing phase** (top-down): The event starts at the root and travels down through ancestors to the target element.
2. **Target phase**: The event reaches the element that triggered it.
3. **Bubbling phase** (bottom-up): The event then bubbles up from the target back to the root, passing through ancestors again.

Most events bubble by default, meaning after the target reacts, its parents get a chance to react too.

10.1.3 Why Propagation Matters

Propagation allows multiple parts of the page to respond to the same event. For example, clicking a button inside a form might trigger the button's event and also the form's event listener.

You can control propagation to prevent unwanted reactions by using methods like `event.stopPropagation()`.

10.1.4 Summary

- Events are actions triggered by user interactions or browser behavior.
- Event listeners watch for specific events on DOM elements.
- Events flow through the DOM in three phases: capturing (down), target, and bubbling (up).
- Understanding event flow helps you write efficient, controlled event handling for interactive web pages.

Think of events as messages traveling through a tree, allowing different parts of your webpage to communicate and react seamlessly!

10.2 `addEventListener`

One of the most common ways to respond to user interactions in JavaScript is by using the **`addEventListener`** method. This method attaches event handlers to DOM elements, allowing your webpage to react to clicks, key presses, mouse movements, and many other events.

10.2.1 Attaching Event Listeners with `addEventListener`

Syntax:

```
element.addEventListener(eventType, callbackFunction);
```

- `eventType` is a string like `'click'`, `'keydown'`, or `'submit'`.
- `callbackFunction` is the function that runs when the event happens.

10.2.2 Example: Listening for a Button Click

```
<button id="myButton">Click Me!</button>
```

```
const button = document.getElementById('myButton');

button.addEventListener('click', () => {
  alert('Button was clicked!');
});
```

When you click the button, the alert pops up.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Button Click Example</title>
</head>
<body>

  <button id="myButton">Click Me!</button>

  <script>
    const button = document.getElementById('myButton');

    button.addEventListener('click', () => {
      alert('Button was clicked!');
    });
  </script>

</body>
</html>
```

10.2.3 Multiple Events on One Element

You can add listeners for different events on the same element:

```
button.addEventListener('mouseenter', () => {
  console.log('Mouse entered button');
});

button.addEventListener('mouseleave', () => {
  console.log('Mouse left button');
});
```

Each event triggers its own callback separately.

10.2.4 Removing Event Listeners

Sometimes, you may want to remove an event listener to stop reacting to an event. To do this, you must use a **named function** instead of an anonymous one:

```
function sayHello() {
  console.log('Hello!');
}

button.addEventListener('click', sayHello);

// Later, remove the listener
button.removeEventListener('click', sayHello);
```

10.2.5 Benefits over Inline Handlers or `onclick`

Multiple Listeners

`addEventListener` allows you to add multiple handlers for the same event on one element. Inline handlers (`onclick`) can only have one.

```
button.onclick = () => console.log('First');
button.onclick = () => console.log('Second');
// Only 'Second' runs

button.addEventListener('click', () => console.log('First'));
button.addEventListener('click', () => console.log('Second'));
// Both run
```

Separation of Concerns

Using `addEventListener` keeps JavaScript out of your HTML, making your code cleaner and easier to maintain.

More Control

`addEventListener` supports options like capturing/bubbling phases, passive listeners, and once-only listeners for better event handling.

10.2.6 Summary

- Use `addEventListener` to attach event handlers dynamically.
- It supports **multiple events** and **multiple handlers** per element.
- You can **remove listeners** using `removeEventListener`.
- It's preferred over inline handlers for cleaner, more flexible code.

By mastering `addEventListener`, you'll gain fine-grained control over user interactions on

your web pages.

10.3 Handling Click, Keyboard, and Form Events

Handling user interactions is at the heart of dynamic web pages. In this section, we'll explore how to respond to **clicks**, **keyboard inputs**, and **form submissions** using event listeners, along with practical tips like preventing default behaviors and validating inputs.

10.3.1 Handling Click Events

Click events are one of the most common ways users interact with a page.

Example – Toggle a class on a button click:

```
<button id="toggleBtn">Toggle Highlight</button>
<div id="box" style="width:100px; height:100px; background-color:lightgray;"></div>
```

```
const button = document.getElementById('toggleBtn');
const box = document.getElementById('box');

button.addEventListener('click', () => {
  box.classList.toggle('highlight');
});
```

```
.highlight {
  background-color: gold;
  border: 2px solid orange;
}
```

Clicking the button toggles the `highlight` style on the box.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Toggle Class Example</title>
  <style>
    #box {
      width: 100px;
      height: 100px;
      background-color: lightgray;
      transition: background-color 0.3s, border 0.3s;
    }

    .highlight {
      background-color: gold;
      border: 2px solid orange;
    }
```

```
  </style>
</head>
<body>

  <button id="toggleBtn">Toggle Highlight</button>
  <div id="box"></div>

  <script>
    const button = document.getElementById('toggleBtn');
    const box = document.getElementById('box');

    button.addEventListener('click', () => {
      box.classList.toggle('highlight');
    });
  </script>

</body>
</html>
```

10.3.2 Handling Keyboard Events

Keyboard events like `keydown` and `keyup` help capture user input from the keyboard.

Example – Detect when the user presses the "Enter" key:

```
document.addEventListener('keydown', (event) => {
  if (event.key === 'Enter') {
    alert('You pressed Enter!');
  }
});
```

Tip: Use `event.key` to detect which key was pressed. You can also handle specific keys like arrows or function keys.

10.3.3 Handling Form Submission and Validation

By default, submitting a form reloads the page. Often, we want to prevent this and validate inputs instead.

Example – Simple form validation:

```
<form id="signupForm">
  <input type="text" id="username" placeholder="Username" />
  <button type="submit">Sign Up</button>
</form>
<p id="message"></p>
```

```
const form = document.getElementById('signupForm');
const usernameInput = document.getElementById('username');
const message = document.getElementById('message');
```

```
form.addEventListener('submit', (event) => {
  event.preventDefault();  // Prevent page reload

  const username = usernameInput.value.trim();

  if (username.length < 3) {
    message.textContent = 'Username must be at least 3 characters long.';
    message.style.color = 'red';
  } else {
    message.textContent = `Welcome, ${username}!`;
    message.style.color = 'green';
    // You can proceed with further processing here
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Simple Form Validation</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }

    input {
      padding: 8px;
      margin-right: 10px;
    }

    button {
      padding: 8px 12px;
    }

    #message {
      margin-top: 10px;
      font-weight: bold;
    }
  </style>
</head>
<body>

  <form id="signupForm">
    <input type="text" id="username" placeholder="Username" />
    <button type="submit">Sign Up</button>
  </form>
  <p id="message"></p>

  <script>
    const form = document.getElementById('signupForm');
    const usernameInput = document.getElementById('username');
    const message = document.getElementById('message');

    form.addEventListener('submit', (event) => {
```

```
      event.preventDefault();  // Prevent page reload

      const username = usernameInput.value.trim();

      if (username.length < 3) {
        message.textContent = 'Username must be at least 3 characters long.';
        message.style.color = 'red';
      } else {
        message.textContent = `Welcome, ${username}!`;
        message.style.color = 'green';
        // You can proceed with further processing here
      }
    });
  </script>

</body>
</html>
```

10.3.4 Practical Tips

- **Use `event.preventDefault()`** to stop default actions (like form submission or link navigation) when you want to handle events yourself.
- Validate inputs before submitting forms to improve user experience.
- When handling keyboard events, prefer `event.key` over deprecated properties like `keyCode`.
- For toggling styles or classes, the `classList.toggle()` method is clean and effective.

10.3.5 Summary

Event Type	Common Use Case	Key Properties/Methods
Click	Button presses, toggling	`click` event, `classList`
Keyboard	Detecting key presses	`keydown`, `keyup`, `event.key`
Form Submit	Validate inputs, prevent reload	`submit`, `event.preventDefault()`

By mastering these event handling techniques, you'll be able to create interactive and user-friendly web pages that respond intuitively to user actions.

10.4 Event Object and Propagation

When an event occurs in the browser, an **event object** is automatically passed to the event handler. This object contains useful information and methods that help you understand and control what happens during the event.

10.4.1 The Event Object: Key Properties and Methods

When you add an event listener, you can access the event object as the first parameter in your callback function:

```
element.addEventListener('click', (event) => {
  // event is the event object
});
```

Here are some important properties and methods:

Property/Method	Description
`event.target`	The element where the event **originated** (clicked, typed, etc.)
`event.currentTarget`	The element that the event listener is currently attached to
`event.preventDefault()`	Stops the browser's default behavior (e.g., link navigation, form submit)
`event.stopPropagation()`	Stops the event from moving further through the DOM (no more bubbling or capturing)

10.4.2 Understanding `target` vs. `currentTarget`

Suppose you have a button inside a container, both with click listeners:

```
<div id="container">
  <button id="btn">Click me</button>
</div>
```

```
const container = document.getElementById('container');
const button = document.getElementById('btn');

container.addEventListener('click', (event) => {
  console.log('container currentTarget:', event.currentTarget.id);
  console.log('container target:', event.target.id);
});

button.addEventListener('click', (event) => {
  console.log('button currentTarget:', event.currentTarget.id);
  console.log('button target:', event.target.id);
});
```

- If you click the button:
  - `event.target` is `btn` (where the event started)
  - `event.currentTarget` is either `btn` or `container` depending on which listener is running.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Event Bubbling Example</title>
  <style>
    #container {
      padding: 20px;
      background-color: lightblue;
    }

    #btn {
      padding: 10px 20px;
    }

    pre {
      background: #f0f0f0;
      padding: 10px;
      margin-top: 20px;
    }
  </style>
</head>
<body>

  <div id="container">
    <button id="btn">Click me</button>
  </div>

  <pre id="log"></pre>

  <script>
    const container = document.getElementById('container');
    const button = document.getElementById('btn');
    const log = document.getElementById('log');

    function logMessage(message) {
      log.textContent += message + '\n';
    }

    container.addEventListener('click', (event) => {
      logMessage('container currentTarget: ' + event.currentTarget.id);
      logMessage('container target: ' + event.target.id);
    });

    button.addEventListener('click', (event) => {
      logMessage('button currentTarget: ' + event.currentTarget.id);
      logMessage('button target: ' + event.target.id);
    });
  </script>
```

```
</body>
</html>
```

10.4.3 Event Propagation: Capturing and Bubbling Phases

When an event happens on a nested element, it flows through the DOM in phases:

1. **Capturing phase**: The event travels *down* from the root (`document`) through ancestors to the target element.
2. **Target phase**: The event reaches the element where it occurred.
3. **Bubbling phase**: The event travels *up* from the target back through ancestors to the root.

Visual analogy: Imagine dropping a pebble in a tree's leaves; the sound travels down branches first (capturing), hits the leaf (target), then echoes up through branches again (bubbling).

10.4.4 Controlling Propagation and Default Behavior

- Use `event.preventDefault()` to **stop default actions** like link navigation or form submission.
- Use `event.stopPropagation()` to **stop the event from bubbling or capturing further**.

10.4.5 Example: Prevent Link Navigation and Stop Bubbling

```
<div id="outer" style="padding:20px; background:#eee;">
  Outer Div
  <a href="https://example.com" id="myLink">Go to Example.com</a>
</div>
```

```
const outer = document.getElementById('outer');
const link = document.getElementById('myLink');

outer.addEventListener('click', () => {
  console.log('Outer div clicked');
});

link.addEventListener('click', (event) => {
  event.preventDefault();        // Prevent link navigation
  event.stopPropagation();       // Stop bubbling up to outer div
  alert('Link clicked but navigation prevented!');
```

```
});
```

Clicking the link will show the alert, **won't navigate away**, and **won't trigger** the outer div's click handler.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Event Propagation & Prevent Default</title>
  <style>
    #outer {
      padding: 20px;
      background: #eee;
      font-family: sans-serif;
    }

    a {
      display: inline-block;
      margin-top: 10px;
      color: blue;
    }
  </style>
</head>
<body>

  <div id="outer">
    Outer Div<br>
    <a href="https://example.com" id="myLink">Go to Example.com</a>
  </div>

  <script>
    const outer = document.getElementById('outer');
    const link = document.getElementById('myLink');

    outer.addEventListener('click', () => {
      console.log('Outer div clicked');
    });

    link.addEventListener('click', (event) => {
      event.preventDefault();        // Prevent link navigation
      event.stopPropagation();       // Stop bubbling up to outer div
      alert('Link clicked but navigation prevented!');
    });
  </script>

</body>
</html>
```

10.4.6 Summary

- The **event object** provides info about the event and lets you control it.
- `target` is where the event started; `currentTarget` is where the handler runs.
- Events flow in three phases: **capturing**, **target**, and **bubbling**.
- Use `preventDefault()` to stop default browser actions.
- Use `stopPropagation()` to stop events from traveling further.

Mastering these tools allows you to create precise and predictable interactive behaviors in your web apps.

Chapter 11.

The Document Object Model (DOM)

1. Understanding the DOM Tree
2. Querying Elements (`getElementById`, `querySelector`)
3. Modifying DOM Elements
4. Creating and Removing DOM Elements

11 The Document Object Model (DOM)

11.1 Understanding the DOM Tree

The **Document Object Model (DOM)** is the way browsers represent a web page internally. It acts as a **tree-like structure** that models every part of an HTML document, allowing JavaScript to access, modify, and interact with the page dynamically.

11.1.1 The DOM as a Tree Structure

Imagine your HTML document as a big **family tree**. At the top is the root, and branches grow down into smaller branches and leaves. Similarly, the DOM tree starts from the `document` root node and branches out into elements like `<html>`, `<body>`, `<div>`, and more.

Each node in this tree represents a part of the document:

- **Element nodes:** These correspond to HTML tags such as `<div>`, `<p>`, `<img>`. They can contain other nodes, including child elements or text.
- **Text nodes:** These represent the text content inside elements. For example, the text inside a paragraph tag is stored as a text node.
- **Other node types:** Like comment nodes and attribute nodes, though elements and text nodes are the most commonly used.

11.1.2 How the Browser Builds the DOM

When a browser loads an HTML page, it reads the HTML code from top to bottom and creates the DOM tree node by node:

```
<html>
  <body>
    <h1>Welcome</h1>
    <p>Hello, world!</p>
  </body>
</html>
```

This HTML creates a DOM tree like this:

```
document
 +- html
     +- body
         +- h1
              +- "Welcome" (text node)
         +- p
              +- "Hello, world!" (text node)
```

11.1.3 Why the Tree Model Matters

This hierarchical tree structure lets JavaScript **navigate** and **manipulate** any part of the page easily. For example, you can find the `<p>` element inside `<body>`, change its text, add new elements, or remove existing ones—all by moving through this tree.

Think of the DOM as the **living blueprint** of your web page—a dynamic representation that your scripts can explore and update in real time.

Understanding the DOM tree is the first step toward mastering how web pages work behind the scenes and interact with your JavaScript code!

11.2 Querying Elements (`getElementById`, `querySelector`)

To interact with a webpage's content, JavaScript needs to **select** or **query** elements from the DOM tree. Two of the most common ways to find elements are using `getElementById` and `querySelector`.

11.2.1 `getElementById`

- **Purpose:** Selects a single element based on its unique `id` attribute.
- **Syntax:** `document.getElementById('elementId')`
- **Returns:** The element object or `null` if no element matches.
- **Performance:** Very fast because IDs are unique.

Example:

```
<div id="main-header">Welcome!</div>
```

```
const header = document.getElementById('main-header');
console.log(header.textContent); // Output: Welcome!
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>textContent Example</title>
  <style>
    #main-header {
      font-size: 24px;
      font-weight: bold;
      margin: 20px;
    }
  </style>
</head>
```

```
<body>

  <div id="main-header">Welcome!</div>

  <script>
    const header = document.getElementById('main-header');
    alert(header.textContent); // Output: Welcome!
  </script>

</body>
</html>
```

11.2.2 querySelector and querySelectorAll

- **Purpose:** Select elements using **CSS selectors**.
- **Syntax:**
  - `document.querySelector(selector)` — returns the **first** matching element.
  - `document.querySelectorAll(selector)` — returns **all** matching elements as a static NodeList.
- **Returns:** Single element (or `null`) for `querySelector`, and a NodeList for `querySelectorAll`.
- **Performance:** Slightly slower than `getElementById` but very flexible.

Examples:

```
<ul>
  <li class="item">Apple</li>
  <li class="item">Banana</li>
  <li class="item">Cherry</li>
</ul>
```

```
const firstItem = document.querySelector('.item');
console.log(firstItem.textContent); // Output: Apple

const allItems = document.querySelectorAll('.item');
allItems.forEach(item => console.log(item.textContent));
// Output:
// Apple
// Banana
// Cherry
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Log to Page Example</title>
  <style>
```

```
    #log {
      background: #f0f0f0;
      padding: 10px;
      font-family: monospace;
      white-space: pre-wrap;
    }
  </style>
</head>
<body>

  <h2>Console Log Output:</h2>
  <pre id="log"></pre>

  <script>
    // Create a log function that prints to both console and the page
    function logToPage(message) {
      console.log(message); // Still logs to browser console
      const logElement = document.getElementById('log');
      logElement.textContent += message + '\n';
    }

    // Example usage
    logToPage('Hello, world!');
    logToPage('Logging Apple, Banana, Cherry:');

    const fruits = ['Apple', 'Banana', 'Cherry'];
    fruits.forEach(fruit => logToPage(fruit));
  </script>

</body>
</html>
```

11.2.3 Differences and When to Use Each

Feature	`getElementById`	`querySelector` / `querySelectorAll`
Selects by	Unique `id`	Any CSS selector
Returns	Single element or `null`	Single element (`querySelector`) or NodeList (`querySelectorAll`)
Use case	Fast, simple ID selection	Flexible, complex selections (classes, tags, attributes, combinations)
Performance	Faster for ID-based queries	Slightly slower but usually negligible

11.2.4 Exercises

1. Select the element with id `"footer"` and change its background color:

```
const footer = document.getElementById('footer');
footer.style.backgroundColor = 'lightblue';
```

2. Select the first button with class `"btn"` and disable it:

```
const firstBtn = document.querySelector('.btn');
firstBtn.disabled = true;
```

3. Select all paragraphs inside a container with id `"content"` and log their text:

```
const paragraphs = document.querySelectorAll('#content p');
paragraphs.forEach(p => console.log(p.textContent));
```

11.2.5 Summary

- Use **`getElementById`** for fast, straightforward access to elements by ID.
- Use **`querySelector`** and **`querySelectorAll`** for flexible, CSS-based selections.
- Both methods are essential tools for DOM manipulation in JavaScript.

Mastering element querying sets the foundation for dynamic web page interactions!

11.3 Modifying DOM Elements

Once you've selected elements from the DOM, the next step is to **modify** them to change how your webpage looks or behaves. You can update an element's content, attributes, styles, or classes—all in real time using JavaScript.

11.3.1 Changing Text Content

To update the visible text inside an element, use:

- `.textContent` — sets or gets the plain text inside an element.
- `.innerHTML` — sets or gets HTML content (including tags) inside an element.

Example:

```
<p id="message">Hello, world!</p>
```

```
const message = document.getElementById('message');

// Change text content (safe, no HTML interpreted)
```

```
message.textContent = 'Welcome to JavaScript!';

// Or update HTML content (can include tags)
message.innerHTML = 'Welcome to <strong>JavaScript</strong>!';
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>textContent vs innerHTML</title>
  <style>
    p {
      font-size: 18px;
      font-family: sans-serif;
    }
  </style>
</head>
<body>

  <p id="message">Hello, world!</p>

  <script>
    const message = document.getElementById('message');

    // Change text content (safe, plain text)
    message.textContent = 'Welcome to JavaScript!';

    // Then update HTML content (can include tags)
    message.innerHTML = 'Welcome to <strong>JavaScript</strong>!';
  </script>

</body>
</html>
```

11.3.2 Modifying Styles

You can change CSS styles directly via the `.style` property.

Example:

```
message.style.color = 'blue';
message.style.fontWeight = 'bold';
message.style.backgroundColor = '#f0f0f0';
```

This immediately updates the paragraph's color, weight, and background.

Full runnable code:

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

```
  <meta charset="UTF-8" />
  <title>Style with JavaScript</title>
  <style>
    #message {
      padding: 10px;
      font-size: 18px;
      font-family: Arial, sans-serif;
    }
  </style>
</head>
<body>

  <p id="message">This text will be styled with JavaScript.</p>

  <script>
    const message = document.getElementById('message');

    message.style.color = 'blue';
    message.style.fontWeight = 'bold';
    message.style.backgroundColor = '#f0f0f0';
  </script>

</body>
</html>
```

11.3.3 Adding, Removing, and Toggling Classes

Classes control CSS styling and can be easily manipulated with the `classList` property:

- `.classList.add('className')` — adds a class
- `.classList.remove('className')` — removes a class
- `.classList.toggle('className')` — toggles a class on/off

Example:

```
message.classList.add('highlight');
message.classList.remove('highlight');
message.classList.toggle('highlight'); // Adds if missing, removes if present
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>classList Example</title>
  <style>
    #message {
      padding: 10px;
      font-size: 18px;
      font-family: Arial, sans-serif;
      background-color: white;
      transition: background-color 0.3s;
```

```
    }

    .highlight {
      background-color: yellow;
      font-weight: bold;
      color: darkblue;
    }
  </style>
</head>
<body>

  <p id="message">This text will be highlighted.</p>
  <button id="toggleBtn">Toggle Highlight</button>

  <script>
    const message = document.getElementById('message');
    const button = document.getElementById('toggleBtn');

    // Add highlight class
    // message.classList.add('highlight');

    // Remove highlight class
    // message.classList.remove('highlight');

    // Toggle highlight class on button click
    button.addEventListener('click', () => {
      message.classList.toggle('highlight');
    });
  </script>

</body>
</html>
```

11.3.4 Modifying Attributes

Attributes like `src` (images), `href` (links), `alt`, or `title` can be changed with:

- `.setAttribute(attributeName, value)`
- `.getAttribute(attributeName)`
- Direct property access (e.g., `element.href = '...'`)

Example:

```
<a id="myLink" href="https://example.com">Visit Example</a>
<img id="myImage" src="image1.jpg" alt="Image">
```

```
const link = document.getElementById('myLink');
const image = document.getElementById('myImage');

link.setAttribute('href', 'https://java2s.com');
link.textContent = 'Visit java2s.com';

image.src = 'image2.jpg';
image.alt = 'New Image';
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Attribute Manipulation Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    img {
      display: block;
      margin-top: 10px;
      max-width: 300px;
      height: auto;
    }
  </style>
</head>
<body>

  <a id="myLink" href="https://example.com" target="_blank">Visit Example</a>
  <img id="myImage" src="https://java2s.com/style/image1.jpg" alt="Image">

  <script>
    const link = document.getElementById('myLink');
    const image = document.getElementById('myImage');

    // Update link href and text
    link.setAttribute('href', 'https://java2s.com');
    link.textContent = 'Visit java2s.com';

    // Update image src and alt attributes
    image.src = 'https://java2s.com/style/image2.jpg';
    image.alt = 'New Image';
  </script>

</body>
</html>
```

11.3.5 Real-Time Changes

All these modifications instantly update the page without needing to reload it. This dynamic behavior is what makes web apps interactive and responsive to user input.

11.3.6 Summary

- Use `.textContent` or `.innerHTML` to change element content.
- Modify styles dynamically via `.style`.
- Use `.classList` methods to manage CSS classes cleanly.
- Change element attributes with `.setAttribute` or direct properties.
- These changes take effect immediately, letting you create rich, interactive pages.

Mastering DOM modifications lets you bring your web pages to life with JavaScript!

11.4 Creating and Removing DOM Elements

Manipulating the DOM isn't limited to modifying existing elements — you can also **create new elements** dynamically and **remove elements** when they are no longer needed. This ability is essential for building interactive, dynamic web pages.

11.4.1 Creating New Elements with `document.createElement`

To create a new element, use the `document.createElement` method:

```
const newDiv = document.createElement('div');
newDiv.textContent = 'Hello, I am a new div!';
```

At this point, `newDiv` exists only in memory — it's not yet visible on the page.

11.4.2 Adding Elements to the DOM

To insert your new element into the document, you attach it to an existing element using methods like:

- `appendChild()` — adds the new element as the last child.
- `insertBefore()` — inserts the new element before a specific existing child.

Example: Add a new item to a list

```
<ul id="todoList">
  <li>Buy groceries</li>
  <li>Walk the dog</li>
</ul>
```

```
const list = document.getElementById('todoList');
const newItem = document.createElement('li');
newItem.textContent = 'Read a book';
```

```
// Add new item to the end of the list
list.appendChild(newItem);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Add List Item Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    ul {
      list-style-type: disc;
      margin-left: 20px;
    }
  </style>
</head>
<body>

  <h2>To-Do List</h2>
  <ul id="todoList">
    <li>Buy groceries</li>
    <li>Walk the dog</li>
  </ul>

  <script>
    const list = document.getElementById('todoList');
    const newItem = document.createElement('li');
    newItem.textContent = 'Read a book';

    // Add new item to the end of the list
    list.appendChild(newItem);
  </script>

</body>
</html>
```

11.4.3 Inserting Before a Specific Element

```
const secondItem = list.children[1]; // The second <li>
const urgentItem = document.createElement('li');
urgentItem.textContent = 'Pay bills';

// Insert urgentItem before secondItem
list.insertBefore(urgentItem, secondItem);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Insert Before Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    ul {
      list-style-type: disc;
      margin-left: 20px;
    }
  </style>
</head>
<body>

  <h2>To-Do List</h2>
  <ul id="todoList">
    <li>Buy groceries</li>
    <li>Walk the dog</li>
  </ul>

  <script>
    const list = document.getElementById('todoList');
    const secondItem = list.children[1]; // The second <li> ("Walk the dog")
    const urgentItem = document.createElement('li');
    urgentItem.textContent = 'Pay bills';

    // Insert urgentItem before secondItem
    list.insertBefore(urgentItem, secondItem);
  </script>

</body>
</html>
```

11.4.4 Removing Elements from the DOM

There are two common ways to remove elements:

1. Using the **parent** element's `removeChild()` method:

```
const firstItem = list.children[0];
list.removeChild(firstItem);
```

2. Using the element's own `remove()` method (modern browsers):

```
const lastItem = list.lastElementChild;
lastItem.remove();
```

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Remove Elements Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    ul {
      list-style-type: disc;
      margin-left: 20px;
    }
    button {
      margin-right: 10px;
      margin-top: 10px;
      padding: 5px 10px;
    }
  </style>
</head>
<body>

  <h2>To-Do List</h2>
  <ul id="todoList">
    <li>Buy groceries</li>
    <li>Walk the dog</li>
    <li>Read a book</li>
  </ul>

  <button id="removeFirst">Remove First Item (removeChild)</button>
  <button id="removeLast">Remove Last Item (remove)</button>

  <script>
    const list = document.getElementById('todoList');
    const removeFirstBtn = document.getElementById('removeFirst');
    const removeLastBtn = document.getElementById('removeLast');

    removeFirstBtn.addEventListener('click', () => {
      const firstItem = list.children[0];
      if (firstItem) {
        list.removeChild(firstItem);
      } else {
        alert('No items left to remove!');
      }
    });

    removeLastBtn.addEventListener('click', () => {
      const lastItem = list.lastElementChild;
      if (lastItem) {
        lastItem.remove();
      } else {
        alert('No items left to remove!');
      }
    });
  </script>

</body>
```

```
</html>
```

11.4.5 Practical Scenario: Dynamic Content Management

Imagine a simple to-do app where users can add and remove tasks dynamically.

```
// Add task
function addTask(text) {
  const li = document.createElement('li');
  li.textContent = text;
  list.appendChild(li);
}

// Remove first task
function removeFirstTask() {
  if (list.firstElementChild) {
    list.firstElementChild.remove();
  }
}

addTask('Learn DOM manipulation');
removeFirstTask();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simple To-Do App</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
      max-width: 400px;
    }
    ul {
      list-style-type: disc;
      margin-left: 20px;
      padding-left: 0;
    }
    li {
      margin: 5px 0;
    }
    input[type="text"] {
      width: 70%;
      padding: 8px;
      font-size: 16px;
    }
    button {
      padding: 8px 12px;
      margin-left: 5px;
      font-size: 16px;
```

```
    }
  </style>
</head>
<body>

  <h2>Simple To-Do App</h2>

  <form id="taskForm">
    <input type="text" id="taskInput" placeholder="Enter new task" required />
    <button type="submit">Add Task</button>
  </form>

  <button id="removeFirstBtn">Remove First Task</button>

  <ul id="taskList"></ul>

  <script>
    const list = document.getElementById('taskList');
    const form = document.getElementById('taskForm');
    const input = document.getElementById('taskInput');
    const removeFirstBtn = document.getElementById('removeFirstBtn');

    // Add task function
    function addTask(text) {
      const li = document.createElement('li');
      li.textContent = text;
      list.appendChild(li);
    }

    // Remove first task function
    function removeFirstTask() {
      if (list.firstElementChild) {
        list.firstElementChild.remove();
      }
    }

    // Form submit handler
    form.addEventListener('submit', event => {
      event.preventDefault(); // prevent page reload
      const taskText = input.value.trim();
      if (taskText) {
        addTask(taskText);
        input.value = '';
        input.focus();
      }
    });

    // Remove first task button
    removeFirstBtn.addEventListener('click', () => {
      removeFirstTask();
    });

    // Initial example task
    addTask('Learn DOM manipulation');
  </script>

</body>
</html>
```

11.4.6 Summary

- Use `document.createElement()` to create new DOM elements.
- Insert them into the document using `appendChild()` or `insertBefore()`.
- Remove elements with `removeChild()` from the parent or the element's own `remove()` method.
- Dynamically adding and removing elements allows you to build interactive web pages that respond in real time.

By mastering creation and removal, you control the structure and content of your web pages dynamically!

Chapter 12.

Error Handling and Debugging

1. Types of Errors (Syntax vs Runtime)
2. Using `try`, `catch`, `finally`
3. The `throw` Statement
4. Debugging with Developer Tools

12 Error Handling and Debugging

12.1 Types of Errors (Syntax vs Runtime)

When writing JavaScript, errors are inevitable, especially as you learn. Understanding the two main types of errors — **syntax errors** and **runtime errors** — helps you quickly identify and fix problems in your code.

12.1.1 Syntax Errors

Syntax errors happen when your code violates the rules of JavaScript's language structure. They are like grammar mistakes in a sentence.

- **Definition:** Errors caused by incorrect code formatting or missing characters.
- **Examples:**

```
let x = 10  // Missing semicolon (optional but good practice)
let y = ;   // Syntax error: unexpected token ";"
function() { console.log("Hi") }  // Missing function name
```

Browsers catch syntax errors **before** running the code, so the script won't execute until you fix them. The browser's developer console usually shows a clear message pointing to the line with the syntax error, like **"Unexpected token"** or **"Missing)"**.

12.1.2 Runtime Errors

Runtime errors occur when your code is syntactically correct but something goes wrong while the program runs.

- **Definition:** Errors caused by invalid operations during execution.
- **Examples:**

Full runnable code:

```
let num = 10;
console.log(num.toUpperCase()); // Runtime error: toUpperCase is not a function

let obj = null;
console.log(obj.name);          // Runtime error: Cannot read property 'name' of null
```

Unlike syntax errors, runtime errors occur **during execution**, which means your script starts but then crashes or behaves unexpectedly. Browsers report runtime errors with messages like **"TypeError"**, **"ReferenceError"**, or **"RangeError"** in the console.

12.1.3 Common Beginner Mistakes

- Missing or extra brackets, parentheses, or semicolons cause syntax errors.
- Calling functions or accessing properties on `undefined` or `null` leads to runtime errors.
- Typos in variable or function names result in **ReferenceErrors** during runtime.

12.1.4 How to Identify Errors

- **Syntax errors** stop your script from running and are flagged immediately.
- **Runtime errors** appear while the script runs, often stopping execution or causing unexpected behavior.
- Always check your browser's **developer console** for error messages and line numbers.

Knowing the difference between syntax and runtime errors is the first step to effective debugging and writing robust JavaScript code!

12.2 Using `try`, `catch`, `finally`

In JavaScript, **error handling** is crucial for building resilient applications that don't crash unexpectedly. The `try`, `catch`, and `finally` blocks allow you to run code safely and handle errors gracefully when they occur.

12.2.1 The `try` Block

The `try` block contains the code that might throw an error during execution. If an error occurs inside `try`, the normal program flow stops and control moves to the `catch` block.

```
try {
  // Code that may cause an error
  let result = riskyOperation();
  console.log(result);
}
```

12.2.2 The `catch` Block

The `catch` block **catches** the error thrown in the `try` block and allows you to handle it without crashing your program.

Full runnable code:

```
try {
  let data = JSON.parse('invalid JSON string');
} catch (error) {
  console.error('Parsing failed:', error.message);
}
```

Here, instead of the script crashing on invalid JSON, the error is caught and logged.

12.2.3 The `finally` Block

The `finally` block runs **after** the `try` and `catch` blocks, **regardless of whether an error occurred or not**. Use it for cleanup actions like closing files or releasing resources.

Full runnable code:

```
try {
  console.log('Trying something risky...');
  // Code that may throw
} catch (error) {
  console.error('Caught an error:', error);
} finally {
  console.log('This always runs, no matter what.');
}
```

12.2.4 Complete Example: Safe Division

Full runnable code:

```
function safeDivide(a, b) {
  try {
    if (b === 0) {
      throw new Error('Cannot divide by zero');
    }
    return a / b;
  } catch (error) {
    console.error('Error:', error.message);
    return null; // Graceful fallback
  } finally {
    console.log('Division attempt finished');
  }
}

console.log(safeDivide(10, 2)); // Outputs: 5
console.log(safeDivide(10, 0)); // Logs error and outputs: null
```

12.2.5 When to Use `try`, `catch`, `finally`

- Use `try` to wrap code that may fail (e.g., parsing data, network requests).
- Use `catch` to handle errors and prevent crashes.
- Use `finally` for cleanup that must happen no matter what, like closing connections or resetting variables.

By using `try`, `catch`, and `finally`, you can make your JavaScript programs more robust, user-friendly, and easier to debug.

12.3 The `throw` Statement

In JavaScript, you can **manually trigger errors** using the `throw` statement. This is useful for signaling exceptional situations in your code that require special handling or to stop execution when something goes wrong.

12.3.1 How `throw` Works

The `throw` statement lets you generate an error of any type, but typically you throw an **Error object** or one of its subclasses (like `TypeError`, `RangeError`). When an error is thrown, the normal flow of the program immediately stops and control jumps to the nearest `catch` block (if one exists).

Syntax:

```
throw expression;
```

12.3.2 Example: Throwing a Custom Error

You can create meaningful errors with clear messages to help debugging and communicate problems:

Full runnable code:

```
function checkAge(age) {
  if (age < 18) {
    throw new Error('User must be at least 18 years old.');
  }
  return 'Access granted';
}

try {
```

```
  console.log(checkAge(16));
} catch (error) {
  console.error('Error:', error.message);
}
```

Here, `throw` creates a new error that stops execution if the age is less than 18. The `catch` block then handles it gracefully.

12.3.3 Throwing Different Error Types

JavaScript has built-in error types you can use for more specific error handling:

```
throw new TypeError('Expected a number but got a string');
throw new RangeError('Value out of allowed range');
```

Using specific error types improves clarity when debugging or logging errors.

12.3.4 Why Use `throw`?

- **Control program flow:** Stop execution immediately when something unexpected happens.
- **Signal exceptional conditions:** Let calling code know that a problem occurred.
- **Improve debugging:** Custom messages help quickly identify where and why an error happened.

12.3.5 Best Practices for `throw`

- Always throw **Error objects** (or subclasses), not strings or other types.
- Provide **clear, descriptive error messages**.
- Use specific error types when applicable.
- Pair `throw` with `try...catch` blocks to handle errors properly.

Manually throwing errors with `throw` empowers you to write safer, more predictable JavaScript by explicitly managing when and how your program handles problems.

12.4 Debugging with Developer Tools

Debugging is an essential skill for every JavaScript developer. Modern browsers like Chrome and Firefox come with powerful **Developer Tools** that help you find and fix issues in your code efficiently. Here's a hands-on guide to get you started with debugging.

12.4.1 Opening Developer Tools

- **Chrome:** Press `F12` or `Ctrl + Shift + I` (Windows/Linux) or `Cmd + Option + I` (Mac).
- **Firefox:** Press `F12` or `Ctrl + Shift + I` (Windows/Linux) or `Cmd + Option + I` (Mac).

This opens a panel with multiple tabs like **Elements**, **Console**, **Sources** (Chrome) or **Debugger** (Firefox).

12.4.2 Viewing Console Errors

The **Console** tab shows errors, warnings, and logs:

- Errors appear in red with messages and the file/line number.
- Click the link to jump to the source of the error.
- Use `console.log()`, `console.error()`, or `console.warn()` in your code to print messages here.

12.4.3 Setting Breakpoints

Breakpoints pause code execution at a specific line so you can inspect the program's state.

1. Open the **Sources** (Chrome) or **Debugger** (Firefox) tab.
2. Navigate to your JavaScript file in the file explorer.
3. Click the line number where you want to pause.

The code will stop when it reaches this line during execution.

12.4.4 Inspecting Variables

When paused on a breakpoint:

- Hover over variables to see their current values.
- Look at the **Scope** or **Local** panel to see all variables available in the current context.
- Use the **Watch** panel to monitor specific variables or expressions.

12.4.5 Stepping Through Code

Use the controls to move through your code step-by-step:

- **Step Over (F10):** Execute the next line, skipping inside function calls.
- **Step Into (F11):** Enter inside a function call.
- **Step Out (Shift + F11):** Exit the current function.
- **Resume (F8):** Continue running until the next breakpoint.

12.4.6 Example: Debugging a Function

Suppose you have this code:

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

let result = add(5, '3');
console.log(result); // Outputs "53" instead of 8
```

- Set a breakpoint on the `return` line.
- Run the code, it pauses there.
- Inspect variables `a` and `b` and see that `b` is a string.
- Realize you need to convert `b` to a number before adding.

12.4.7 Tips for Effective Debugging

- Use breakpoints instead of `alert()` for cleaner debugging.
- Check the **Call Stack** panel to see how you arrived at the current code line.
- Combine debugging with console logs for quick checks.
- Refresh the page with DevTools open to catch errors on load.

Mastering Developer Tools gives you powerful insight into your JavaScript code, helping you find bugs faster and write better software!

Chapter 13.

Working with Dates and Times

1. Creating and Formatting Dates
2. Timestamps and Time Differences
3. Timers: `setTimeout`, `setInterval`

13 Working with Dates and Times

13.1 Creating and Formatting Dates

Working with dates and times is a common task in JavaScript, and the `Date` object provides a powerful way to handle this.

13.1.1 Creating Date Objects

You can create a `Date` object in several ways:

- **Current date and time:**

Full runnable code:

```
const now = new Date();
console.log(now); // Outputs current date and time, e.g. 2025-06-23T14:30:00.000Z
```

- **Specific date/time using a date string:**

Full runnable code:

```
const specificDate = new Date('2025-12-25T10:00:00');
console.log(specificDate); // Dec 25, 2025, 10:00 AM
```

- **Using individual date and time components:**

Full runnable code:

```
const customDate = new Date(2025, 11, 25, 10, 0, 0);
// Note: Month is zero-based, so 11 = December
console.log(customDate);
```

13.1.2 Formatting Dates

JavaScript provides several built-in methods to convert Date objects into readable strings:

- **`toLocaleDateString()`** Formats the date according to the local conventions:

Full runnable code:

```
const now = new Date();
console.log(now.toLocaleDateString()); // e.g. "6/23/2025" in US locale
console.log(now.toLocaleDateString('en-GB')); // "23/06/2025" for UK
```

- **`toISOString()`** Returns the date in ISO 8601 format — great for APIs:

Full runnable code:

```
const now = new Date();
console.log(now.toISOString()); // "2025-06-23T14:30:00.000Z"
```

13.1.3 Custom Formatting with Getters

You can also extract specific parts of the date and build your own format:

- `.getFullYear()` — Year (e.g., 2025)
- `.getMonth()` — Month (0–11, so add 1 for human-readable)
- `.getDate()` — Day of the month
- `.getHours()`, `.getMinutes()`, `.getSeconds()` — Time parts

Example:

Full runnable code:

```
const now = new Date();
const year = now.getFullYear();
const month = now.getMonth() + 1; // Add 1 because months start at 0
const day = now.getDate();
const hours = now.getHours();
const minutes = now.getMinutes();

const formatted = `${day}/${month}/${year} ${hours}:${minutes}`;
console.log(formatted); // e.g. "23/6/2025 14:30"
```

13.1.4 Practical Example: Formatting the Current Date

Full runnable code:

```
const currentDate = new Date();

console.log('Locale date:', currentDate.toLocaleDateString());
console.log('ISO string:', currentDate.toISOString());
console.log('Custom format:', `${currentDate.getDate()}/${currentDate.getMonth() + 1}/${currentDate.getl
```

This outputs the current date in multiple human- and machine-friendly formats, helping you display or process dates in the way your application needs.

Mastering date creation and formatting unlocks powerful capabilities for handling time-based data in JavaScript!

13.2 Timestamps and Time Differences

When working with dates and times in JavaScript, understanding **timestamps** and how to calculate differences between dates is crucial for many applications — such as measuring elapsed time, creating countdowns, or even calculating someone's age.

13.2.1 What is a Timestamp?

A **timestamp** is a way of representing a specific moment in time as a single number. In JavaScript, timestamps are expressed as the number of **milliseconds** that have passed since the **Unix epoch**, which is January 1, 1970, 00:00:00 UTC. This format allows for precise time calculations and comparisons.

13.2.2 Getting the Current Timestamp

You can get the current timestamp using:

- `Date.now()` — returns the number of milliseconds since the Unix epoch for the current moment.
- `dateObject.getTime()` — returns the timestamp for a specific `Date` object.

Full runnable code:

```
// Using Date.now()
const nowTimestamp = Date.now();
console.log(nowTimestamp);  // e.g., 1687551937264

// Using getTime() on a Date object
const now = new Date();
const timestampFromDate = now.getTime();
console.log(timestampFromDate);  // Should be the same as Date.now()
```

13.2.3 Calculating Time Differences

To find the difference between two dates, you subtract their timestamps. The result is the difference in milliseconds, which you can then convert to seconds, minutes, hours, or days.

Example: Calculate Days Between Two Dates

Full runnable code:

```
const date1 = new Date('2025-07-01');
const date2 = new Date('2025-07-10');

const diffInMs = date2.getTime() - date1.getTime();  // Difference in milliseconds
const millisecondsPerDay = 1000 * 60 * 60 * 24;      // 86,400,000 ms in a day
const diffInDays = diffInMs / millisecondsPerDay;

console.log(`Days between: ${diffInDays}`);  // Outputs: Days between: 9
```

13.2.4 Real-World Examples

Measuring Elapsed Time

Suppose you want to measure how long a task takes in your program:

Full runnable code:

```
const startTime = Date.now();

// Simulate some process with a delay
setTimeout(() => {
  const endTime = Date.now();
  const elapsedTimeMs = endTime - startTime;
  console.log(`Elapsed time: ${elapsedTimeMs} milliseconds`);
}, 2000);
```

This code measures the time elapsed during a 2-second delay.

Countdown Timer

You can calculate the remaining time until a future event by subtracting the current timestamp from the event's timestamp.

Full runnable code:

```
const futureDate = new Date('2025-12-31T23:59:59');
const now = Date.now();

const timeLeftMs = futureDate.getTime() - now;

if (timeLeftMs > 0) {
  const secondsLeft = Math.floor(timeLeftMs / 1000);
  console.log(`Countdown: ${secondsLeft} seconds left`);
} else {
  console.log("The event has already occurred!");
}
```

Calculating Age from a Birthdate

You can calculate someone's age by comparing their birthdate to the current date:

Full runnable code:

```
function calculateAge(birthDateStr) {
  const birthDate = new Date(birthDateStr);
  const now = new Date();

  const diffInMs = now.getTime() - birthDate.getTime();
  const millisecondsPerYear = 1000 * 60 * 60 * 24 * 365.25; // Approximate, accounts for leap years

  return Math.floor(diffInMs / millisecondsPerYear);
}

const age = calculateAge('1990-06-23');
console.log(`Age: ${age} years`);
```

13.2.5 Summary

- **Timestamps** represent points in time as milliseconds since January 1, 1970.
- Use `Date.now()` or `getTime()` to get timestamps.
- Subtract timestamps to find durations between dates.
- Convert milliseconds to desired units like seconds, minutes, or days.
- Use these concepts to build useful features like elapsed timers, countdowns, or age calculations.

Mastering timestamps and date differences opens up many possibilities for handling time in JavaScript!

13.3 Timers: `setTimeout`, `setInterval`

JavaScript provides two essential timer functions — `setTimeout` and `setInterval` — that allow you to schedule code execution **after a delay** or **repeatedly at intervals**. These are incredibly useful for creating dynamic behaviors like delayed messages, animations, countdowns, and polling.

13.3.1 `setTimeout`: Delayed Execution

`setTimeout` schedules a function to run **once** after a specified number of milliseconds.

Basic Syntax

```
const timeoutId = setTimeout(functionToRun, delayInMilliseconds);
```

- `functionToRun`: The callback function to execute.
- `delayInMilliseconds`: How long to wait before running the function.

Example: Show a Message After 3 Seconds

Full runnable code:

```
setTimeout(() => {
  console.log("This message appears after 3 seconds.");
}, 3000);
```

After 3 seconds, the message will appear in the console.

13.3.2 Canceling a Timeout

If you need to cancel a scheduled timeout before it runs, save its ID and call `clearTimeout()`:

```
const timeoutId = setTimeout(() => {
  console.log("You won't see this message.");
}, 5000);

// Cancel the timeout before it runs
clearTimeout(timeoutId);
```

13.3.3 `setInterval`: Repeated Execution

`setInterval` runs a function **repeatedly** at fixed intervals (in milliseconds) until stopped.

Basic Syntax

```
const intervalId = setInterval(functionToRun, intervalInMilliseconds);
```

- `functionToRun`: The callback function to execute repeatedly.
- `intervalInMilliseconds`: How often to run the function.

Example: Counter Increment Every Second

Full runnable code:

```
let count = 0;

const intervalId = setInterval(() => {
  count++;
  console.log(`Count: ${count}`);

  if (count >= 5) {
    clearInterval(intervalId);  // Stop after 5 counts
```

```
    console.log("Counter stopped.");
  }
}, 1000);
```

This code increments and logs the counter every second. After reaching 5, it stops the interval.

13.3.4 Canceling an Interval

Use `clearInterval()` with the interval ID to stop repeated execution:

```
clearInterval(intervalId);
```

13.3.5 Best Practices & Common Pitfalls

- **Avoid overlapping intervals:** If your interval callback takes longer than the interval duration, callbacks can pile up, causing unexpected behavior. To prevent this, you can:
  - Use `setTimeout` recursively instead of `setInterval` when the callback involves asynchronous or longer tasks.
  - Clear the interval and restart it inside the callback.
- **Always clear timers:** To prevent memory leaks or unwanted behavior, clear timeouts and intervals when they are no longer needed (e.g., when a component unmounts in frameworks or when the user navigates away).
- **Use meaningful variable names:** Naming your timeout or interval IDs clearly (`messageTimeout`, `counterInterval`) helps keep your code readable.

13.3.6 Practical Example: Delayed Greeting and Repeated Reminder

Full runnable code:

```
// Show a greeting after 2 seconds
const greetingTimeout = setTimeout(() => {
  console.log("Hello! Welcome to our site.");
}, 2000);

// Every 3 seconds, remind the user to check notifications
let reminderCount = 0;
const reminderInterval = setInterval(() => {
  reminderCount++;
  console.log("Don't forget to check your notifications!");
```

```
  if (reminderCount === 3) {
    clearInterval(reminderInterval);
    console.log("Reminder stopped after 3 times.");
  }
}, 3000);
```

13.3.7 Summary

Function	Purpose	Cancel with
`setTimeout`	Run code once after a delay	`clearTimeout()`
`setInterval`	Run code repeatedly at intervals	`clearInterval()`

Timers are simple yet powerful tools to control timing in your JavaScript programs. Use them wisely to create smooth user experiences, animations, and automated tasks!

Chapter 14.

Strings and String Methods

1. Common String Methods (`slice`, `substr`, `replace`, `split`, `trim`)
2. Template Literals
3. String Searching and Pattern Matching

14 Strings and String Methods

14.1 Common String Methods (`slice`, `substr`, `replace`, `split`, `trim`)

JavaScript strings come with many built-in methods that help you manipulate text easily. In this section, we'll explore some of the most commonly used string methods:

- `slice()`
- `substr()`
- `replace()`
- `split()`
- `trim()`

Each method serves a specific purpose, such as extracting parts of a string, replacing content, splitting strings into arrays, or removing unwanted whitespace.

14.1.1 `slice()`

The `slice()` method extracts a part of a string and returns it as a new string without modifying the original.

Syntax:

```
string.slice(startIndex, endIndex)
```

- `startIndex` (required): The position where extraction begins (zero-based).
- `endIndex` (optional): The position where extraction ends (not included). If omitted, extracts till the end.

Example:

Full runnable code:

```
const fullName = "Jane Doe";

// Extract first name (characters from index 0 to 4)
const firstName = fullName.slice(0, 4);
console.log(firstName);  // Output: "Jane"

// Extract last name (from index 5 to end)
const lastName = fullName.slice(5);
console.log(lastName);   // Output: "Doe"
```

14.1.2 substr()

The `substr()` method extracts a substring starting from a given index and for a specified length.

Syntax:

```
string.substr(startIndex, length)
```

- `startIndex` (required): The starting position.
- `length` (optional): Number of characters to extract.

Example:

Full runnable code:

```
const url = "https://example.com/page";

// Extract "example"
const domain = url.substr(8, 7);
console.log(domain);  // Output: "example"
```

> **Note:** `substr()` is considered a legacy method and may not be supported in all environments. `slice()` or `substring()` are generally preferred.

14.1.3 replace()

The `replace()` method replaces part of a string with another string. It returns a new string without changing the original.

Syntax:

```
string.replace(searchValue, newValue)
```

- `searchValue`: The substring or regular expression to find.
- `newValue`: The string to replace the found substring.

Example:

Full runnable code:

```
const greeting = "Hello, world!";
const newGreeting = greeting.replace("world", "JavaScript");
console.log(newGreeting);  // Output: "Hello, JavaScript!"
```

> By default, `replace()` only changes the first match. To replace all occurrences, use a regular expression with the global flag `/g`:

Full runnable code:

```
const sentence = "Apples are red. Apples are sweet.";
const updated = sentence.replace(/Apples/g, "Oranges");
console.log(updated);
// Output: "Oranges are red. Oranges are sweet."
```

14.1.4 split()

The `split()` method divides a string into an array of substrings based on a specified separator.

Syntax:

```
string.split(separator, limit)
```

- `separator`: The character(s) or regular expression to split the string on.
- `limit` (optional): Maximum number of splits.

Example:

Full runnable code:

```
const sentence = "JavaScript,Python,Ruby,Java";

// Split by commas
const languages = sentence.split(",");
console.log(languages);  // Output: ["JavaScript", "Python", "Ruby", "Java"]

// Split into maximum 2 parts
const limitedSplit = sentence.split(",", 2);
console.log(limitedSplit);  // Output: ["JavaScript", "Python"]
```

14.1.5 trim()

The `trim()` method removes whitespace from both ends of a string but **not** from inside the string.

Syntax:

```
string.trim()
```

Example:

Full runnable code:

```
const rawInput = "   user@example.com   ";
const cleanedInput = rawInput.trim();
console.log(cleanedInput);  // Output: "user@example.com"
```

This is especially useful when processing user input to avoid errors caused by accidental

spaces.

14.1.6 Summary

Method	Purpose	Key Parameters	Example Use Case
`slice`	Extract substring by start/end	`startIndex`, `endIndex`	Get first or last name
`substr`	Extract substring by start/length	`startIndex`, `length`	Extract part of URL
`replace`	Replace substring	`searchValue`, `newValue`	Replace words in sentences
`split`	Split string into array	`separator`, `limit`	Parse CSV or list strings
`trim`	Remove whitespace around string	None	Clean user input

By mastering these common string methods, you'll be able to manipulate and transform text efficiently in your JavaScript programs!

14.2 Template Literals

Template literals are a modern and powerful way to work with strings in JavaScript. Introduced in ES6, they offer a more readable and flexible alternative to traditional string concatenation.

14.2.1 What Are Template Literals?

Template literals are strings enclosed by backticks (`` ` ``) instead of single (') or double (") quotes.

```
const message = `Hello, world!`;
```

Using backticks lets you easily:

- **Embed variables and expressions** inside strings
- Create **multi-line strings** without special characters
- Write **more readable and maintainable code**

14.2.2 Benefits Over Traditional Concatenation

Instead of this:

```
const name = "Alice";
const age = 25;

const greeting = "Hello, " + name + "! You are " + age + " years old.";
console.log(greeting);
```

With template literals, you can write:

Full runnable code:

```
const name = "Alice";
const age = 25;

const greeting = `Hello, ${name}! You are ${age} years old.`;
console.log(greeting);
```

The second version is clearer, shorter, and easier to maintain.

14.2.3 Variable Interpolation with {}

Inside a template literal, anything placed inside `${}` is evaluated as JavaScript code and its result is inserted into the string.

Full runnable code:

```
const price = 9.99;
const quantity = 3;

console.log(`Total price: $${price * quantity}`);  // Output: Total price: $29.97
```

You can embed variables, mathematical expressions, function calls, or any valid JavaScript expression.

14.2.4 Multi-line Strings

With traditional strings, to create a multi-line string, you had to use escape characters `\n` or concatenate strings:

```
const poem = "Roses are red,\nViolets are blue,\nJavaScript is fun,\nAnd so are you.";
```

With template literals, you can simply write:

Full runnable code:

```
const poem = `Roses are red,
Violets are blue,
JavaScript is fun,
And so are you.`;

console.log(poem);
```

This preserves line breaks and formatting exactly as written.

14.2.5 Mini-Project: Dynamic Greeting and HTML Snippet Generator

Let's create a small example that generates a personalized greeting and an HTML snippet using template literals.

Full runnable code:

```
function createGreeting(name, dayOfWeek) {
  return `
    <div class="greeting">
      <h1>Hello, ${name}!</h1>
      <p>Hope you're having a great ${dayOfWeek}.</p>
      <p>Today is ${new Date().toLocaleDateString()}.</p>
    </div>
  `;
}

const userName = "Sam";
const today = "Tuesday";

const greetingHTML = createGreeting(userName, today);
console.log(greetingHTML);
```

Output:

```
<div class="greeting">
  <h1>Hello, Sam!</h1>
  <p>Hope you're having a great Tuesday.</p>
  <p>Today is 6/23/2025.</p>
</div>
```

This HTML string can be injected into a webpage dynamically, demonstrating how template literals make string building easier and cleaner.

14.2.6 Summary

- Template literals use backticks ‘ to define strings.
- Use `${}` to embed variables and expressions inside strings.
- Support multi-line strings naturally without extra characters.

- Improve code readability and reduce errors compared to traditional concatenation.
- Perfect for generating dynamic content such as greetings, emails, or HTML snippets.

Try replacing your old concatenation code with template literals — your future self will thank you!

14.3 String Searching and Pattern Matching

Finding and matching specific parts of a string is a common task in JavaScript. This section introduces simple methods for searching strings as well as an introduction to **regular expressions (RegExp)** for more powerful pattern matching.

14.3.1 Simple String Searching Methods

JavaScript provides several built-in methods to search within strings:

Method	Description	Returns
`indexOf()`	Finds the index of a substring	Index number or -1 if not found
`includes()`	Checks if a substring exists	`true` or `false`
`startsWith()`	Checks if string begins with a substring	`true` or `false`
`endsWith()`	Checks if string ends with a substring	`true` or `false`

14.3.2 `indexOf()`

Returns the **position** of the first occurrence of a substring. If not found, returns `-1`.

Full runnable code:

```
const text = "JavaScript is fun";

console.log(text.indexOf("Script"));  // Output: 4
console.log(text.indexOf("python"));  // Output: -1 (not found)
```

14.3.3 `includes()`

Returns `true` if the substring exists anywhere in the string, otherwise `false`.

Full runnable code:

```
const email = "user@example.com";

console.log(email.includes("@"));       // Output: true
console.log(email.includes("gmail"));   // Output: false
```

14.3.4 `startsWith()` and `endsWith()`

Check if a string starts or ends with a specified substring.

Full runnable code:

```
const url = "https://java2s.com";

console.log(url.startsWith("https"));  // true
console.log(url.endsWith(".com"));     // true
console.log(url.startsWith("http"));   // true
console.log(url.endsWith(".org"));      // false
```

14.3.5 Introduction to Regular Expressions (RegExp)

When simple substring searches are not enough, **regular expressions** provide a flexible way to match complex patterns in strings.

A **RegExp** is a pattern used to match character combinations.

You create regex patterns with:

```
const pattern = /pattern/flags;
```

or

```
const pattern = new RegExp("pattern", "flags");
```

14.3.6 Using RegExp for Searches

Example: Check if a string contains only digits

Full runnable code:

```
const input = "12345";
const digitPattern = /^\d+$/;  // ^ = start, \d = digit, + = one or more, $ = end
```

```
console.log(digitPattern.test(input));  // true
console.log(digitPattern.test("123a5")); // false
```

14.3.7 Practical Use Case: Validate an Email Format

Here is a simple regex to validate a basic email format:

Full runnable code:

```
const email = "user@example.com";
const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if (emailPattern.test(email)) {
  console.log("Valid email address.");
} else {
  console.log("Invalid email address.");
}
```

- `^` and `$` mark the start and end of the string.
- `[^\s@]+` means one or more characters that are **not** whitespace or `@`.
- `@` is matched literally.
- `\.` matches the dot before the domain extension.

14.3.8 Searching With RegExp Methods

You can also use `.match()` to find matches:

Full runnable code:

```
const sentence = "I have 2 cats and 3 dogs.";
const numbers = sentence.match(/\d+/g);  // Find all numbers

console.log(numbers);  // Output: ["2", "3"]
```

- `/\d+/g` searches for one or more digits globally (`g` flag).
- `.match()` returns an array of matches.

14.3.9 Summary

Method	Purpose	Returns
`indexOf()`	Position of substring	Number or -1

Method	Purpose	Returns
`includes()`	Check if substring exists	Boolean
`startsWith()`	Check prefix of string	Boolean
`endsWith()`	Check suffix of string	Boolean
`RegExp.test()`	Test if string matches pattern	Boolean
`String.match()`	Find matches for regex	Array of matches or null

14.3.10 Final Note

For simple substring searches, `indexOf`, `includes`, `startsWith`, and `endsWith` are straightforward and efficient. For more advanced pattern matching and validation — such as email formats, phone numbers, or custom rules — regular expressions are an invaluable tool.

With practice, you'll gain the ability to harness both simple methods and regex to build robust string handling in your JavaScript programs!

Chapter 15.

Object-Oriented Programming (OOP) in JavaScript

1. Constructor Functions
2. Prototypes and Inheritance
3. ES6 Classes
4. Encapsulation with Closures and Private Fields

15 Object-Oriented Programming (OOP) in JavaScript

15.1 Constructor Functions

In JavaScript, **constructor functions** are a traditional way to create multiple objects with similar properties and methods. They provide a blueprint to produce many instances, each with their own unique data.

15.1.1 What Is a Constructor Function?

A constructor function is simply a regular function that is used with the `new` keyword to create new objects. By convention, constructor function names start with a capital letter to distinguish them from normal functions.

15.1.2 How Does `this` Work Inside a Constructor?

Inside a constructor function, `this` refers to the **new object** being created. When you call a constructor with `new`, JavaScript automatically:

1. Creates a new empty object.
2. Sets `this` to point to that new object.
3. Runs the constructor function code.
4. Returns the new object (unless the constructor explicitly returns something else).

15.1.3 Defining a Constructor Function

Here's a simple example defining a `Person` constructor:

```
function Person(name, age) {
  this.name = name;  // Assign name property to the new object
  this.age = age;    // Assign age property to the new object

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

15.1.4 Creating Multiple Instances

Using the `new` keyword, you can create as many `Person` objects as you want, each with its own name and age:

```
const alice = new Person("Alice", 30);
const bob = new Person("Bob", 25);

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

Each instance has its own copy of the `name`, `age`, and the `greet` method.

15.1.5 Example: `Car` Constructor

Let's look at another example—a `Car` constructor function:

Full runnable code:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;

  this.getInfo = function() {
    return `${this.year} ${this.make} ${this.model}`;
  };
}

const car1 = new Car("Toyota", "Corolla", 2020);
const car2 = new Car("Tesla", "Model 3", 2023);

console.log(car1.getInfo());  // Output: 2020 Toyota Corolla
console.log(car2.getInfo());  // Output: 2023 Tesla Model 3
```

15.1.6 Summary

- Constructor functions allow you to create multiple objects with shared structure.
- The `this` keyword inside a constructor refers to the new object being created.
- Use `new` to instantiate objects from a constructor function.
- Each instance has its own properties and methods, enabling flexible and reusable code.

Constructor functions form the foundation of object creation in JavaScript and are essential to understand before moving on to more advanced OOP concepts like prototypes and ES6 classes.

15.2 Prototypes and Inheritance

JavaScript uses a **prototype-based inheritance** model, which is quite different from the classical inheritance found in many other programming languages. Understanding prototypes is key to writing efficient, reusable object-oriented code in JavaScript.

15.2.1 What Is a Prototype?

Every JavaScript object has an internal link to another object called its **prototype**. This prototype object can have properties and methods that the original object can access indirectly. This chain of links is called the **prototype chain**.

15.2.2 Why Use Prototypes?

When you create multiple objects using a constructor function, you usually want to share common methods among them. If you define methods inside the constructor, each instance gets its own copy — which wastes memory.

Instead, JavaScript allows you to add shared methods to the constructor's **prototype**. All instances then refer to the same method, saving memory and improving performance.

15.2.3 Example: Prototype Sharing

Full runnable code:

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

// Adding a method to the prototype, shared by all instances
Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}.`);
};

const alice = new Person("Alice", 30);
const bob = new Person("Bob", 25);

alice.greet();  // Hello, my name is Alice.
bob.greet();    // Hello, my name is Bob.

// Checking prototype relationship
console.log(alice.__proto__ === Person.prototype);  // true
```

Here, `greet` is defined only once on `Person.prototype`, but both `alice` and `bob` can use it.

15.2.4 The Prototype Chain

If you try to access a property or method on an object, JavaScript first looks at the object itself. If it doesn't find it there, it looks up the prototype chain until it finds it or reaches the end (`null`).

```
console.log(alice.hasOwnProperty("name"));       // true (own property)
console.log(alice.hasOwnProperty("greet"));      // false (in prototype)
console.log(typeof alice.greet);                  // "function" (found via prototype)
```

15.2.5 Inheritance via Prototypes

You can create inheritance relationships between constructors by linking their prototypes.

Suppose we want a `Student` to inherit from `Person`:

Full runnable code:

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

// Adding a method to the prototype, shared by all instances
Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}.`);
};


function Student(name, age, grade) {
  Person.call(this, name, age);  // Call Person constructor with Student's context
  this.grade = grade;
}

// Set Student's prototype to inherit from Person.prototype
Student.prototype = Object.create(Person.prototype);

// Fix constructor pointer because it points to Person now
Student.prototype.constructor = Student;

// Add a method specific to Student
Student.prototype.study = function() {
  console.log(`${this.name} is studying.`);
};

const student1 = new Student("Charlie", 20, "A");
student1.greet();   // Inherited method: Hello, my name is Charlie.
```

```
student1.study();   // Student method: Charlie is studying.
```

In this example:

- `Student` calls `Person` constructor to initialize shared properties.
- `Student.prototype` is linked to `Person.prototype` via `Object.create()`.
- This sets up the prototype chain so `student1` inherits `greet()` from `Person`.
- Additional `Student`-specific methods like `study()` are added to `Student.prototype`.

15.2.6 Summary

- JavaScript objects inherit properties and methods through a **prototype chain**.
- Methods shared via a constructor's `.prototype` save memory by avoiding copies on each instance.
- Use `Object.create()` to set up inheritance between constructor prototypes.
- Constructor functions can call parent constructors to initialize inherited properties.
- Understanding prototypes is fundamental for efficient and powerful JavaScript OOP.

Prototypes unlock the power of JavaScript inheritance, enabling code reuse and memory efficiency in your programs!

15.3 ES6 Classes

With ES6, JavaScript introduced the **class syntax** — a cleaner and more intuitive way to write constructor functions and work with inheritance. Although classes are mostly **syntactic sugar** over the existing prototype system, they make object-oriented programming easier to read and write.

15.3.1 Class Declaration and Constructor

A class declaration defines a blueprint for creating objects. Inside the class, the `constructor` method sets up new instances.

Full runnable code:

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

```
  // Method shared by all instances
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}

const alice = new Person("Alice", 30);
alice.greet();  // Output: Hello, my name is Alice.
```

- The `constructor` method runs automatically when you create a new instance with `new`.
- Methods like `greet` are added to the prototype behind the scenes.

15.3.2 Inheritance with `extends` and `super`

Classes can extend other classes to inherit properties and methods.

- The `extends` keyword sets up the inheritance relationship.
- The `super()` function calls the parent class's constructor.

Full runnable code:

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

  // Method shared by all instances
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}


class Student extends Person {
  constructor(name, age, grade) {
    super(name, age);  // Call Person's constructor
    this.grade = grade;
  }

  // Override greet method
  greet() {
    console.log(`Hi, I am ${this.name}, and my grade is ${this.grade}.`);
  }

  study() {
    console.log(`${this.name} is studying.`);
  }
}

const student1 = new Student("Bob", 22, "A");
```

```
student1.greet();   // Hi, I am Bob, and my grade is A.
student1.study();   // Bob is studying.
```

- `Student` inherits `name` and `age` from `Person`.
- The `greet` method is **overridden** to provide custom behavior.
- The `study` method is specific to `Student`.

15.3.3 Summary

Feature	Description	Example
Class Declaration	Defines blueprint with `class` keyword	`class Person { ... }`
Constructor	Initializes new instances	`constructor(name, age) { ... }`
Methods	Defined inside class, shared via prototype	`greet() { ... }`
Inheritance	Use `extends` to inherit a class	`class Student extends Person`
Super Call	Call parent constructor with `super()`	`super(name, age)`
Method Overriding	Redefine parent method in subclass	`greet() { ... }`

15.3.4 Why Use ES6 Classes?

- **Clearer syntax** — easier to understand at a glance.
- **Better organization** — groups data and behavior naturally.
- **Standardized inheritance** — straightforward subclassing and method overriding.
- **Compatibility** — under the hood, classes use prototypes, so performance and flexibility remain.

15.3.5 Final Example: Using Classes

Full runnable code:

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

```
  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog("Rex");
dog.speak();  // Rex barks.
```

This simple example demonstrates how subclasses can extend and customize parent class behavior, all with a clean and concise syntax.

With ES6 classes, JavaScript's object-oriented programming becomes more accessible and expressive — giving you a powerful tool for building structured and maintainable code!

15.4 Encapsulation with Closures and Private Fields

Encapsulation is a fundamental concept in object-oriented programming that restricts direct access to some of an object's components, protecting the internal state and exposing only what's necessary. In JavaScript, encapsulation can be achieved using **closures** and, more recently, **private class fields**.

15.4.1 Encapsulation with Closures

Before private fields were introduced, one common way to create private variables and methods was through **closures** in constructor or factory functions. Variables defined inside the constructor or factory function are not accessible outside, but can be used by privileged methods.

Example: Private Variables Using Closures in a Constructor Function

Full runnable code:

```
function BankAccount(initialBalance) {
  let balance = initialBalance;  // Private variable

  // Privileged method has access to private variable
  this.deposit = function(amount) {
    if (amount > 0) {
      balance += amount;
```

```
      console.log(`Deposited: $${amount}`);
    }
  };

  this.getBalance = function() {
    return balance;
  };
}

const account = new BankAccount(100);
account.deposit(50);
console.log(account.getBalance());  // Output: 150

// Trying to access balance directly will fail
console.log(account.balance);  // undefined
```

- `balance` is private because it's scoped inside the constructor and inaccessible from outside.
- Only `deposit` and `getBalance` methods can access or modify `balance`.

15.4.2 Encapsulation with Factory Functions and Closures

Similarly, you can use factory functions to create objects with private data:

Full runnable code:

```
function createCounter() {
  let count = 0;  // private variable

  return {
    increment() {
      count++;
    },
    getCount() {
      return count;
    }
  };
}

const counter = createCounter();
counter.increment();
console.log(counter.getCount());  // 1
console.log(counter.count);       // undefined
```

15.4.3 Private Class Fields (`#fieldName`)

Starting with ES2022, JavaScript introduced **private class fields** and methods using a `#` prefix. These fields are truly private and cannot be accessed or modified from outside the

class.

Example: Using Private Fields in a Class

Full runnable code:

```
class User {
  #password;  // Private field

  constructor(username, password) {
    this.username = username;
    this.#password = password;
  }

  checkPassword(input) {
    return input === this.#password;
  }

  // Private method
  #encryptPassword() {
    // pretend encryption logic here
    return btoa(this.#password);
  }

  showEncryptedPassword() {
    return this.#encryptPassword();
  }
}

const user = new User("alice", "secret123");
console.log(user.username);               // alice
console.log(user.checkPassword("secret123"));  // true

// Trying to access private field or method from outside causes error
//console.log(user.#password);  // SyntaxError
//console.log(user.#encryptPassword());  // SyntaxError
```

- Private fields and methods are declared with `#` before the name.
- They cannot be accessed or changed outside the class, enforcing strong encapsulation.

15.4.4 Why Encapsulation Matters

- **Improves maintainability:** By hiding internal details, you can change implementation without affecting external code.
- **Increases security:** Prevents accidental or malicious modification of important data.
- **Encourages cleaner APIs:** Expose only necessary properties and methods, making objects easier to understand and use.

15.4.5 Summary

Technique	How It Works	Accessibility
Closures	Variables scoped inside functions	Private to constructor/factory functions
Private Class Fields (`#`)	Fields declared with `#` inside classes	Truly private, inaccessible outside class

Encapsulation in JavaScript protects your data and helps build robust, secure, and maintainable applications. Whether using closures or modern private fields, it's an essential part of writing good object-oriented code.

Chapter 16.

Asynchronous JavaScript

1. Synchronous vs Asynchronous Execution
2. Callbacks
3. Promises
4. `async`/`await`
5. Error Handling in Async Code

16 Asynchronous JavaScript

16.1 Synchronous vs Asynchronous Execution

Understanding the difference between **synchronous** and **asynchronous** execution is key to mastering JavaScript programming.

16.1.1 What Is Synchronous Execution?

Synchronous code runs **one line at a time**, in order. Each operation must finish before the next one starts — much like standing in a single line at a coffee shop. You wait patiently for the person ahead of you to finish before it's your turn.

In JavaScript, this means the program executes statements sequentially, blocking further code from running until the current operation completes.

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

The output will always be:

```
Step 1
Step 2
Step 3
```

16.1.2 What Is Asynchronous Execution?

Asynchronous code, on the other hand, allows your program to **start a task and move on to others before that task finishes** — like cooking dinner while waiting for laundry to finish. You don't just stand idle; you multitask.

In JavaScript, this means some operations (like fetching data from the internet or reading files) happen in the background. Your program can continue running without waiting, and when the background task finishes, a callback function handles the result.

16.1.3 Why Asynchronous JavaScript?

JavaScript runs in a single thread, so synchronous long-running tasks (like network requests) would freeze the program and make the user interface unresponsive. Asynchronous programming prevents this by allowing these operations to happen without blocking other

code.

16.1.4 The Event Loop and Callbacks (Brief Introduction)

JavaScript uses an **event loop** to manage asynchronous operations. When an async task completes, its callback function is queued and executed once the main code finishes running. This allows your program to handle multiple tasks efficiently without freezing.

Example:

```
console.log("Start");

setTimeout(() => {
  console.log("Async task done");
}, 1000);

console.log("End");
```

Output:

```
Start
End
Async task done
```

Here, `setTimeout` schedules the callback to run after 1 second, but the program continues executing and logs "End" immediately.

16.1.5 Summary

- **Synchronous:** Code runs line by line, blocking the next operation until done.
- **Asynchronous:** Tasks run in the background, allowing other code to execute without waiting.
- Asynchronous JavaScript improves performance and user experience by avoiding freezes.
- The **event loop** manages asynchronous callbacks, enabling multitasking in a single-threaded environment.

In the next sections, we'll explore how JavaScript implements asynchronous programming using callbacks, promises, and the modern `async`/`await` syntax.

16.2 Callbacks

Callbacks are one of the fundamental building blocks of asynchronous JavaScript. A **callback** is simply a function that is passed as an argument to another function and executed later, usually once some task completes.

16.2.1 How Callbacks Work

In JavaScript, functions are **first-class objects**, meaning they can be passed around just like any other value. When you pass a function as an argument, the receiving function can call it whenever appropriate.

Basic Example

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

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("Alice", sayGoodbye);
```

Output:

```
Hello, Alice!
Goodbye!
```

Here, `sayGoodbye` is passed as a callback to `greet` and executed after the greeting.

16.2.2 Callbacks in Asynchronous Tasks

Callbacks are especially useful for asynchronous operations like timers, reading files, or handling events.

Example: Using `setTimeout`

```
console.log("Start");

setTimeout(() => {
  console.log("This happens after 2 seconds");
}, 2000);

console.log("End");
```

Output:

```
Start
End
This happens after 2 seconds
```

The anonymous function passed to `setTimeout` is the callback that runs after the delay.

16.2.3 Callbacks in Event Handling

When responding to user actions, you often pass callbacks to event listeners:

```
document.getElementById("btn").addEventListener("click", () => {
  alert("Button clicked!");
});
```

The function you provide is called when the button is clicked.

16.2.4 Callback Hell: When Things Get Messy

While callbacks are powerful, nesting them too deeply leads to **callback hell** — code that is hard to read, maintain, and debug.

Example of Callback Hell

```
loginUser(username, password, (error, user) => {
  if (error) {
    console.error(error);
  } else {
    fetchUserProfile(user.id, (error, profile) => {
      if (error) {
        console.error(error);
      } else {
        updateUI(profile, (error) => {
          if (error) {
            console.error(error);
          } else {
            console.log("UI updated successfully");
          }
        });
      }
    });
  }
});
```

This pyramid-like structure is difficult to follow and leads to bugs and frustration.

16.2.5 Why Callbacks Alone Are Not Enough

- **Readability suffers** with nested callbacks.
- **Error handling** becomes complex.
- **Inversion of control:** You lose direct control over the flow, making debugging harder.

Because of these issues, JavaScript introduced better abstractions like **Promises** and `async/await`, which simplify asynchronous code.

16.2.6 Summary

- Callbacks are functions passed as arguments and executed later.
- They are essential for handling asynchronous operations like timers and events.
- Excessive nested callbacks lead to "callback hell," making code hard to maintain.
- Promises and async/await provide cleaner alternatives to callbacks, which we will explore next.

Callbacks are the foundation of async JavaScript — mastering them will help you understand how asynchronous flow works before moving to modern patterns.

16.3 Promises

Promises are a modern way to handle asynchronous operations in JavaScript, designed to improve upon the limitations of callbacks. They make asynchronous code easier to read, write, and maintain.

16.3.1 What Is a Promise?

A **Promise** represents the eventual result of an asynchronous operation. It's like a placeholder that promises to deliver a value in the future — or an error if something goes wrong.

16.3.2 Promise States

A Promise can be in one of three states:

- **Pending:** The initial state; the operation is still in progress.
- **Fulfilled:** The operation completed successfully, and a result is available.

- **Rejected:** The operation failed with an error.

Once a Promise settles (fulfilled or rejected), its state cannot change.

16.3.3 Creating a Promise

You create a Promise using the `Promise` constructor, which takes a function with two parameters: `resolve` (for success) and `reject` (for failure).

Full runnable code:

```
const delay = (ms) => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Waited for ${ms} milliseconds`);
    }, ms);
  });
};

delay(2000).then((message) => {
  console.log(message);
});
```

In this example, the promise waits for 2 seconds, then resolves with a message.

16.3.4 Using `.then()`, `.catch()`, and `.finally()`

- `.then()` runs when the Promise is fulfilled and receives the resolved value.
- `.catch()` runs if the Promise is rejected, handling errors.
- `.finally()` runs after the Promise settles, regardless of outcome, useful for cleanup.

Full runnable code:

```
const delay = (ms) => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Waited for ${ms} milliseconds`);
    }, ms);
  });
};

delay(1000)
  .then((msg) => {
    console.log(msg);  // Waited for 1000 milliseconds
    throw new Error("Oops!");  // Simulate an error
  })
  .catch((error) => {
    console.error("Error:", error.message);
```

```
  })
  .finally(() => {
    console.log("Done waiting.");
  });
```

16.3.5 Chaining Promises

One of the biggest advantages of Promises is chaining, which lets you perform a sequence of asynchronous operations clearly:

```
fetch("https://api.example.com/data")
  .then((response) => response.json())
  .then((data) => {
    console.log("Data received:", data);
    return delay(1500);  // Wait before next step
  })
  .then(() => {
    console.log("Ready for next operation.");
  })
  .catch((error) => {
    console.error("Fetch error:", error);
  });
```

Here, each `.then()` returns either a value or another Promise, making the flow easy to follow.

16.3.6 Summary

- Promises represent asynchronous results with states: pending, fulfilled, rejected.
- Use `.then()` to handle success, `.catch()` for errors, and `.finally()` for cleanup.
- Promises avoid "callback hell" by enabling clear chaining of async operations.
- They provide a more manageable and readable way to write asynchronous code.

Next, we'll see how the `async/await` syntax builds on Promises to make asynchronous code look even more like synchronous code!

16.4 `async/await`

The `async/await` syntax is a modern, cleaner way to write asynchronous JavaScript code. It builds on Promises and allows you to write asynchronous operations that look and behave like synchronous code, making it easier to read, write, and debug.

16.4.1 What Are `async` and `await`?

- **`async`** is a keyword used to declare a function as asynchronous. This means the function **always returns a Promise**, even if you don't explicitly return one.
- **`await`** can only be used inside `async` functions. It pauses the execution of the function until the awaited Promise is resolved or rejected.

16.4.2 Basic Usage

Here's an example that fetches data using Promises and `async`/`await`:

Full runnable code:

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data fetched!");
    }, 2000);
  });
}

// Using async/await
async function getData() {
  console.log("Fetching data...");
  const result = await fetchData();
  console.log(result);
}

getData();
```

Output:

```
Fetching data...
(Data appears after 2 seconds)
Data fetched!
```

The `await fetchData()` line pauses the function until the Promise resolves, then continues with the result.

16.4.3 Comparison With Promises

Using Promises, the same code looks like this:

```
fetchData().then(result => {
  console.log(result);
});
```

While Promises work fine, chaining multiple asynchronous steps with `.then()` can get messy. With `async/await`, you write code in a **linear, synchronous style** that's easier to follow, especially when dealing with several async calls.

16.4.4 Handling Errors with `try/catch`

One of the biggest advantages of `async/await` is simplified error handling using traditional `try/catch` blocks:

```
async function getDataWithError() {
  try {
    const result = await fetchDataThatMightFail();
    console.log(result);
  } catch (error) {
    console.error("Error occurred:", error);
  }
}
```

This is cleaner than `.catch()` methods on Promises, especially with complex async flows.

16.4.5 Example: Multiple Awaited Calls

Full runnable code:

```
async function processTasks() {
  console.log("Task 1 start");
  await delay(1000);
  console.log("Task 1 complete");

  console.log("Task 2 start");
  await delay(1500);
  console.log("Task 2 complete");
}

function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

processTasks();
```

The output shows tasks running in sequence, with the function pausing at each `await`:

```
Task 1 start
(Task 1 waits 1 second)
Task 1 complete
Task 2 start
(Task 2 waits 1.5 seconds)
```

```
Task 2 complete
```

16.4.6 Summary

- Use `async` to declare a function that returns a Promise.
- Use `await` to pause execution until a Promise resolves or rejects.
- `async`/`await` makes asynchronous code easier to read, write, and debug compared to chained Promises.
- Use `try/catch` for straightforward error handling in async functions.

In the next section, we'll dive deeper into how to handle errors effectively in asynchronous code to write robust, reliable JavaScript applications.

16.5 Error Handling in Async Code

Handling errors effectively in asynchronous JavaScript is essential to building reliable and maintainable applications. Errors can occur during network requests, file operations, or any async task, so knowing how to catch and respond to them is crucial.

16.5.1 Handling Errors with Promises: `.catch()`

When working with Promises, you handle errors using the `.catch()` method, which catches any rejection or thrown error in the Promise chain.

Example:

```
fetch("https://api.example.com/data")
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log("Data received:", data);
  })
  .catch((error) => {
    console.error("Fetch error:", error.message);
  });
```

- If the fetch fails or the server returns an error status, the `.catch()` block handles it.
- Always include `.catch()` at the end of Promise chains to avoid unhandled rejections.

16.5.2 Common Pitfalls with Promises

- Forgetting to return Promises inside `.then()` handlers can break chaining and error propagation.
- Omitting `.catch()` can lead to uncaught promise rejections, causing runtime warnings or crashes.
- Errors thrown **inside** synchronous code inside a `.then()` block are caught by the `.catch()` that follows.

16.5.3 Handling Errors with `async/await`: `try/catch`

When using `async/await`, error handling becomes more intuitive with `try/catch` blocks, similar to synchronous code.

Example:

```
async function getUserData() {
  try {
    const response = await fetch("https://api.example.com/user");
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json();
    console.log("User data:", data);
  } catch (error) {
    console.error("Error fetching user data:", error.message);
  }
}

getUserData();
```

- Any error thrown inside the `try` block—including rejected Promises—is caught by the `catch`.
- This approach results in cleaner, more readable error handling compared to `.then()/.catch()` chains.

16.5.4 Best Practices for Robust Async Error Handling

- Always **handle errors** in every async operation—never ignore them.
- Use `.catch()` with Promises or `try/catch` with `async/await`.
- When possible, add meaningful error messages and context to help with debugging.
- Consider fallback logic or retries when handling recoverable errors.
- For multiple parallel Promises, use `Promise.allSettled()` to handle both successes and failures gracefully.

16.5.5 Summary

Pattern	How to Handle Errors
Promises	Use `.catch()` to catch rejections
`async/await`	Use `try/catch` blocks

Proper error handling is a vital part of asynchronous programming in JavaScript. By catching and managing errors effectively, your applications will be more stable, easier to debug, and provide a better user experience.

Chapter 17.

Working with APIs and Fetch

1. What is an API?
2. Using `fetch` to Make HTTP Requests
3. Reading JSON Responses
4. Building a Simple Weather App

17 Working with APIs and Fetch

17.1 What is an API?

An **API**, or **Application Programming Interface**, is like a bridge that allows two different software systems to talk to each other. It defines a set of rules and protocols that enable programs to request and exchange information in a structured way.

17.1.1 Imagine a Restaurant Analogy

Think of an API as a waiter in a restaurant. You (the customer) don't go into the kitchen to prepare your food; instead, you place an order with the waiter. The waiter takes your request to the kitchen, which prepares the dish, and then the waiter brings it back to you. In this analogy:

- You = The client (your app or program)
- Kitchen = The server or service providing data
- Waiter = The API facilitating communication

You don't need to know how the kitchen prepares the food; you just trust the waiter to bring the correct dish.

17.1.2 What Do APIs Do?

APIs allow your app to interact with other software or services, often over the internet. They let you:

- Request data (like weather forecasts, news, or social media posts).
- Send data (like posting a message or uploading a photo).
- Trigger actions remotely (like turning on a smart light or sending an email).

APIs handle the details of how the data is formatted and transferred, so developers can focus on building the app's features.

17.1.3 Why Are APIs Essential in Modern Web Development?

Most websites and applications rely on data and services provided by others. For example:

- **Weather apps** use APIs to fetch the latest weather data from a weather service.
- **Social media platforms** offer APIs so third-party apps can post updates or read user

profiles.
- **Payment processors** provide APIs for securely handling online transactions.

Without APIs, integrating these external services would be difficult and time-consuming.

17.1.4 Summary

APIs are the invisible helpers that connect different software systems. They simplify communication, allow data sharing, and enable developers to create powerful apps that leverage existing services — making the web more connected and interactive than ever before.

17.2 Using `fetch` to Make HTTP Requests

The `fetch` API is a modern, built-in way to make HTTP requests in JavaScript. It allows your web app to communicate with servers and retrieve or send data over the internet. Let's explore how to use `fetch` to perform a basic GET request, handle responses, and manage errors.

17.2.1 Basic GET Request with `fetch`

A GET request is used to **retrieve data** from a server. Here's how you make a simple GET request using `fetch`:

Full runnable code:

```
fetch('https://api.agify.io?name=michael')
  .then(response => response.json())  // Parse JSON from the response
  .then(data => {
    console.log(JSON.stringify(data, null,2));  // Log the fetched data
  })
  .catch(error => {
    console.error('Error fetching data:', error);
  });
```

17.2.2 How This Works

1. **Calling `fetch()`**: The `fetch` function takes the URL of the resource you want to access and returns a Promise.

2. **Handling the Response**: When the request completes, the Promise resolves to a `Response` object.
3. **Parsing JSON**: Since most APIs send data as JSON, you call `.json()` on the `Response` object, which also returns a Promise that resolves to the JavaScript object.
4. **Using the Data**: Inside the next `.then()`, you work with the parsed data.
5. **Handling Errors**: The `.catch()` block handles any network errors or issues with fetching the data.

17.2.3 A Complete Runnable Example

Try this example in your browser's console or a JavaScript environment that supports `fetch`:

Full runnable code:

```
fetch('https://api.agify.io?name=sarah')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json();
  })
  .then(data => {
    console.log(`Predicted age for Sarah: ${data.age}`);
  })
  .catch(error => {
    console.error('Failed to fetch:', error);
  });
```

17.2.4 Important Notes

- The `fetch` API **does not reject the Promise** on HTTP errors like 404 or 500; you need to check `response.ok` to handle such cases manually.
- The `.json()` method parses the response body as JSON asynchronously.
- Always include `.catch()` to handle network failures or other unexpected errors.

17.2.5 Summary

- Use `fetch(url)` to initiate a network request; it returns a Promise.
- Check `response.ok` to confirm the HTTP status is successful.
- Parse the response with `.json()` to convert it into usable JavaScript objects.
- Handle errors with `.catch()` to ensure your app stays robust.

Next, we'll look at how to read and work with JSON responses in more detail.

17.3 Reading JSON Responses

When working with APIs, the most common data format you'll encounter is **JSON** (JavaScript Object Notation). JSON is a lightweight, human-readable format for exchanging data between servers and clients.

17.3.1 What is JSON?

JSON looks like JavaScript objects but is actually a string format that represents data as key-value pairs, arrays, and nested structures. For example:

```
{
  "name": "Alice",
  "age": 28,
  "hobbies": ["reading", "traveling", "coding"]
}
```

This JSON represents a person's profile with a name, age, and a list of hobbies.

17.3.2 Parsing JSON with `.json()`

When you use the `fetch` API, the response body is received as a stream of bytes. To convert it into a usable JavaScript object, you call the `.json()` method on the response. This method **parses the JSON string** and returns a Promise that resolves with the resulting object.

17.3.3 Example: Fetching and Parsing JSON

Full runnable code:

```
fetch('https://api.agify.io?name=emma')
  .then(response => response.json())   // Parse JSON data
  .then(data => {
    console.log(JSON.stringify(data,null,2));  // Output the whole object
    console.log(`Name: ${data.name}`); // Access individual properties
    console.log(`Predicted Age: ${data.age}`);
  })
  .catch(error => {
    console.error('Error:', error);
```

```
  });
```

Sample output:

```
{ name: "emma", age: 29, count: 12345 }
Name: emma
Predicted Age: 29
```

17.3.4 Working with JSON Arrays

Sometimes, APIs return arrays of objects. Here's an example of working with such data:

Full runnable code:

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => response.json())
  .then(users => {
    users.forEach(user => {
      console.log(`User: ${user.name}, Email: ${user.email}`);
    });
  })
  .catch(error => {
    console.error('Failed to load users:', error);
  });
```

This fetches an array of users and logs each user's name and email.

17.3.5 Summary

- JSON is the most popular format for API responses because it's easy to read and work with.
- Use the `.json()` method on a fetch response to parse JSON into JavaScript objects or arrays.
- Once parsed, access JSON data using standard object or array syntax.
- Properly handle errors to ensure your app can deal with unexpected issues gracefully.

In the next section, we'll put all this together by building a simple weather app that fetches real data from a public API!

17.4 Building a Simple Weather App

In this section, we'll build a simple weather app that fetches live weather data from a public API, parses the JSON response, and dynamically updates the webpage with the information. We'll also add error handling to ensure the app works smoothly.

17.4.1 Step 1: Get API Access

We'll use the OpenWeatherMap API, which provides free weather data. You need to sign up for a free API key.

For this example, let's assume you have an API key stored in a variable:

```
const apiKey = 'YOUR_API_KEY_HERE';
```

17.4.2 Step 2: HTML Setup

Create a simple HTML structure with an input box to enter a city name, a button to fetch weather, and an area to display results:

```
<input type="text" id="cityInput" placeholder="Enter city name" />
<button id="getWeatherBtn">Get Weather</button>

<div id="weatherResult"></div>
```

17.4.3 Step 3: Fetch Weather Data

Write a function to fetch weather data from the API using the city name:

```
const apiKey = 'YOUR_API_KEY_HERE';  // Replace with your actual API key

async function fetchWeather(city) {
  const url = `https://api.openweathermap.org/data/2.5/weather?q=${encodeURIComponent(city)}&appid=${api

  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error('City not found');
    }

    const data = await response.json();
    return data;
  } catch (error) {
    throw error;  // Propagate error to caller
```

```
  }
}
```

- We add `units=metric` to get temperature in Celsius.
- The function throws an error if the city is not found or there's a network issue.

17.4.4 Step 4: Update the DOM Dynamically

Next, create a function to display weather information or errors on the page:

```
function displayWeather(data) {
  const weatherDiv = document.getElementById('weatherResult');
  weatherDiv.innerHTML = `
    <h2>Weather in ${data.name}</h2>
    <p>Temperature: ${data.main.temp} °C</p>
    <p>Conditions: ${data.weather[0].description}</p>
    <p>Humidity: ${data.main.humidity}%</p>
  `;
}

function displayError(message) {
  const weatherDiv = document.getElementById('weatherResult');
  weatherDiv.innerHTML = `<p style="color: red;">Error: ${message}</p>`;
}
```

17.4.5 Step 5: Connect UI and Fetch Logic

Add an event listener to the button to get the city input, fetch the weather, and update the page:

```
document.getElementById('getWeatherBtn').addEventListener('click', async () => {
  const city = document.getElementById('cityInput').value.trim();

  if (!city) {
    displayError('Please enter a city name');
    return;
  }

  displayError('');  // Clear previous errors
  document.getElementById('weatherResult').textContent = 'Loading...';

  try {
    const weatherData = await fetchWeather(city);
    displayWeather(weatherData);
  } catch (error) {
    displayError(error.message);
  }
});
```

17.4.6 Complete Flow

1. User enters a city name and clicks the button.
2. The app fetches live weather data using the OpenWeatherMap API.
3. The JSON response is parsed.
4. Weather details are shown dynamically on the page.
5. Errors like invalid city names or network problems are handled gracefully.

17.4.7 Summary

You've now created a working weather app that demonstrates:

- Making API requests with `fetch` and `async`/`await`.
- Parsing JSON responses.
- Dynamically updating the DOM.
- Handling errors for a smooth user experience.

This simple project is a great example of how JavaScript interacts with APIs to build real-world applications!

Chapter 18.

Modules and Code Organization

1. Why Use Modules?
2. ES6 Modules (`import` / `export`)
3. CommonJS (for Node.js)
4. Organizing Large Projects

18 Modules and Code Organization

18.1 Why Use Modules?

As your JavaScript projects grow larger and more complex, managing all your code in a single file quickly becomes difficult. Imagine trying to organize an entire house in just one big room — it would be messy, confusing, and hard to find anything. This is where **modules** come in, acting like separate rooms that keep everything organized and manageable.

18.1.1 Benefits of Modular Code

1. Maintainability: Modules break your code into smaller, focused pieces, each responsible for a specific task or feature. This makes it easier to understand, debug, and update parts of your application without affecting everything else.

2. Reusability: Just like building blocks, modules can be reused across different projects or parts of the same project. Instead of rewriting code, you can import the modules you need, saving time and reducing errors.

3. Scope Management: In large scripts, variables and functions declared globally can clash or accidentally overwrite each other, leading to bugs. Modules create their own **scope**, so variables inside a module don't pollute the global scope and don't interfere with other parts of the program.

18.1.2 Common Problems Without Modules

- **Code Duplication:** Without modules, you may end up copying and pasting code, making updates tedious and error-prone.
- **Naming Conflicts:** Variables or functions with the same names in different parts of the program can overwrite each other.
- **Difficult Collaboration:** Multiple developers working on one large file can easily cause merge conflicts and confusion.

18.1.3 Modules: The Separate Rooms Analogy

Think of your project as a house. Instead of throwing all your furniture into one room, you divide it into separate rooms: kitchen, bedroom, living room — each with its own purpose and contents. This way, you know exactly where things belong, and you can easily add, remove, or change a room without disrupting the whole house.

18.1.4 Summary

Using modules is essential for writing clean, organized, and scalable JavaScript code. They help you avoid common pitfalls in large scripts and make your codebase easier to maintain, reuse, and collaborate on. In the following sections, we'll explore how to use JavaScript's module systems to build better applications.

18.2 ES6 Modules (`import` / `export`)

ES6 introduced a standardized way to organize JavaScript code using **modules**, allowing you to split your code into separate files and easily share variables, functions, or classes between them. This helps keep your code clean and maintainable.

18.2.1 Exporting from a Module

You can **export** code elements from a file to make them accessible elsewhere. There are two main types of exports:

Named Exports

Export multiple variables, functions, or classes by name:

```
// mathUtils.js
export const PI = 3.14159;

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

export class Calculator {
  multiply(a, b) {
    return a * b;
  }
}
```

Here, `PI`, `add`, and `Calculator` are exported by name.

Default Export

You can export one default value from a module, useful when the module only exports a single item:

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

18.2.2 Importing from a Module

You can **import** exports from other files to use them:

Import Named Exports

```
import { PI, add, Calculator } from './mathUtils.js';

console.log(PI);            // 3.14159
console.log(add(2, 3));     // 5

const calc = new Calculator();
console.log(calc.multiply(4, 5));  // 20
```

You must use the exact exported names when importing named exports.

Import Default Export

```
import greet from './greeting.js';

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

You can name the imported default export anything you like.

Import Both Named and Default

If a module has both, you can import them together:

```
import greet, { PI } from './greetingAndMath.js';
```

18.2.3 Using ES6 Modules in Browsers and Bundlers

- **Modern browsers** support ES6 modules directly using the `<script>` tag with `type="module"`:

```
<script type="module" src="main.js"></script>
```

- Modules loaded this way are **deferred by default**, so they execute after the HTML is parsed.
- For older browsers or more complex projects, bundlers like **Webpack**, **Rollup**, or **Parcel** bundle modules into a single file for compatibility.

18.2.4 Summary

- Use `export` to share variables, functions, and classes from a module.
- Use `import` to bring those exports into other files.

- Named exports allow multiple exports per file; default exports allow one primary export.
- Modern browsers and bundlers fully support ES6 modules, making modular JavaScript easier than ever.

Next, we'll explore how CommonJS modules work in Node.js environments.

18.3 CommonJS (for Node.js)

Before ES6 modules became widely supported, Node.js introduced its own module system called **CommonJS** to organize and reuse code. Even today, CommonJS remains the standard for most Node.js projects.

18.3.1 How CommonJS Works

In CommonJS, each file is treated as a separate module with its own scope. To **export** values (like functions, objects, or variables) from a module, you assign them to `module.exports`. To **import** these exports into another file, you use the `require()` function.

18.3.2 Exporting with `module.exports`

Here's a simple module that exports a function and a variable:

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

const greeting = "Welcome to CommonJS";

module.exports = {
  sayHello,
  greeting,
};
```

18.3.3 Importing with `require()`

You can import the exported members in another file like this:

```
// app.js
const greet = require('./greet');
```

```
console.log(greet.greeting);          // Welcome to CommonJS
console.log(greet.sayHello('Alice')); // Hello, Alice!
```

18.3.4 Differences Between CommonJS and ES6 Modules

Feature	CommonJS	ES6 Modules
Syntax	`require()` / `module.exports`	`import` / `export`
Loading	Synchronous (at runtime)	Asynchronous / static (compile time)
Default in Node.js	Yes	Supported in modern Node.js with `.mjs` extension or config
Browser support	No (needs bundlers)	Supported natively in modern browsers

18.3.5 When to Use Each

- **CommonJS** is the default and widely used in Node.js environments, especially for backend development.
- **ES6 modules** are preferred for modern frontend development and are increasingly supported in Node.js (with configuration).

18.3.6 Summary

CommonJS uses `module.exports` to expose code and `require()` to import it. It enables modular coding in Node.js projects and differs from ES6 modules mainly in syntax and loading behavior. Understanding both systems is essential for working across different JavaScript environments.

Next, we'll explore strategies for organizing larger projects using these modules effectively.

18.4 Organizing Large Projects

As your JavaScript projects grow beyond a few files, organizing your code becomes crucial for maintainability, collaboration, and scalability. This section covers practical strategies to

keep your codebase clean, modular, and easy to navigate.

18.4.1 Folder Structures and Separation of Concerns

A clear folder structure helps separate different parts of your application by their roles or features. Here's a common example:

```
/src
  /components   // Reusable UI components or modules
  /utils        // Utility functions/helpers
  /services     // API calls or external data handling
  /styles       // CSS or styling files
  index.js      // Main entry point
```

- **Separation of concerns** means each module or file has a focused responsibility.
- For example, API logic goes in `/services`, UI logic in `/components`, and helper functions in `/utils`.
- This structure makes it easier to find and update specific parts without digging through huge files.

18.4.2 Naming Conventions

Consistent naming improves readability and reduces confusion:

- Use **camelCase** for variables and functions: `fetchData()`, `userName`
- Use **PascalCase** for classes and React components: `UserProfile`, `WeatherApp`
- Name files clearly and descriptively, often matching the exported class or function (e.g., `userProfile.js` for `UserProfile`).

18.4.3 Using Modules Effectively

- **Break your code into small, reusable modules**. Avoid giant files with many responsibilities.
- Each module should export what's necessary and hide internal details.
- Import only what you need to keep dependencies clear and minimal.
- Modular code encourages easier testing and reusability.

18.4.4 Tools and Techniques

- **Bundlers:** Tools like **Webpack**, **Rollup**, and **Parcel** bundle your modules into optimized files for browsers. They support features like code splitting, tree shaking, and asset management.
- **Linters:** Use linters like **ESLint** to enforce coding standards, catch errors early, and maintain consistent style.
- **Formatters:** Tools like **Prettier** help automatically format your code to a consistent style.
- **Version Control:** Use **Git** or similar systems to track changes and collaborate efficiently.

18.4.5 Summary

Organizing large JavaScript projects requires:

- A clear folder structure based on functionality.
- Consistent naming conventions.
- Writing modular code with well-defined responsibilities.
- Using bundlers and linters to maintain quality and optimize delivery.

Good organization reduces bugs, speeds up development, and makes your codebase welcoming for you and your team. As you grow as a developer, these habits will help you tackle complex projects with confidence.