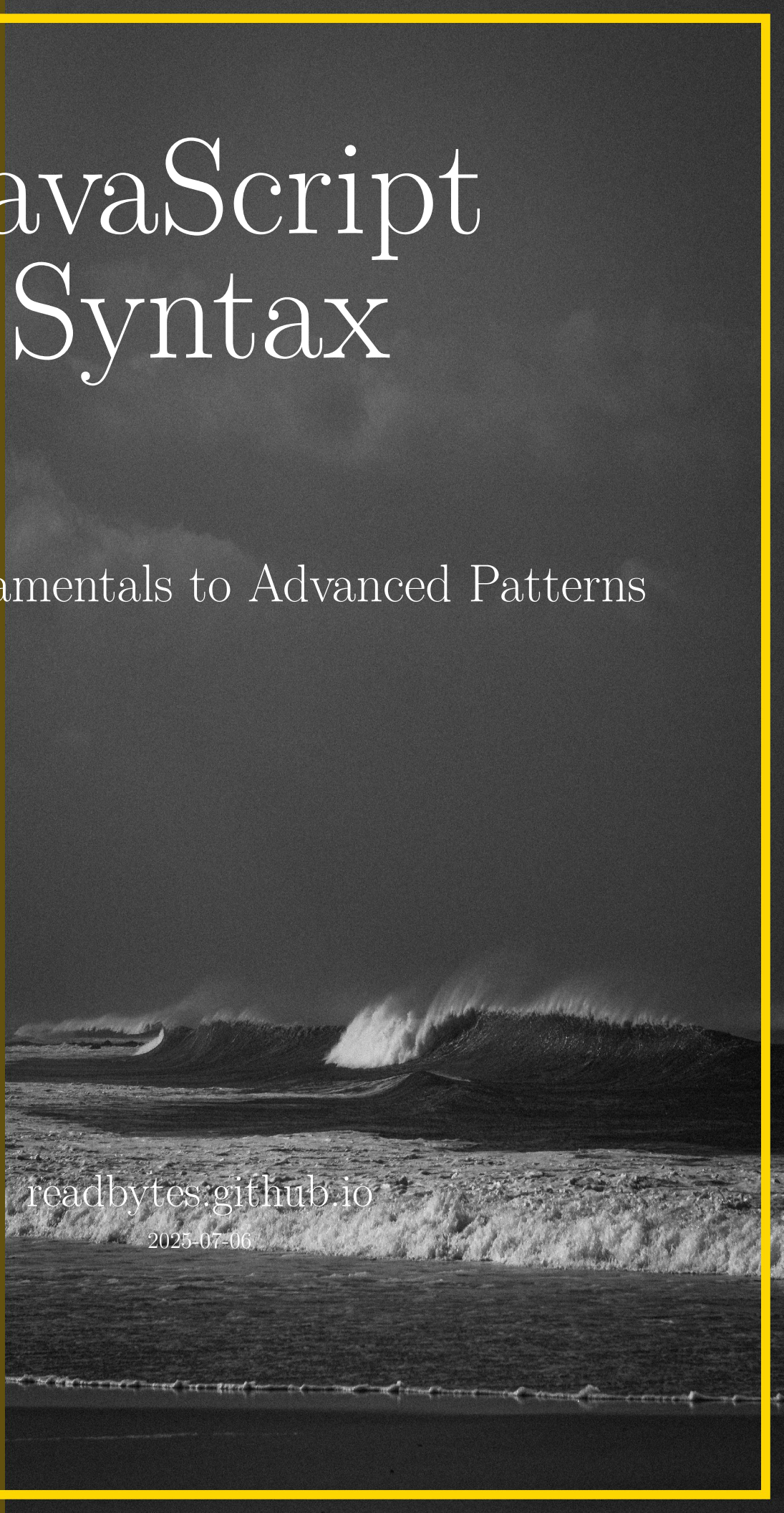
JavaScript Syntax

From Fundamentals to Advanced Patterns



readbytes.github.io

2025-07-06

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to JavaScript

1. What is JavaScript?
2. Setting Up Your Environment
3. Writing Your First JavaScript Program
4. Using the Console and Debugger

1 Introduction to JavaScript

1.1 What is JavaScript?

JavaScript is a versatile, high-level programming language primarily used to create interactive behavior on websites. It is one of the core technologies of the World Wide Web, alongside **HTML** and **CSS**. While HTML provides the structure of a web page, and CSS handles its visual presentation, JavaScript enables dynamic functionality—such as responding to user actions, updating content without reloading the page, validating forms, creating animations, and much more.

JavaScript was originally developed in 1995 by Brendan Eich while working at Netscape. Initially called *Mocha*, then *LiveScript*, it was eventually renamed JavaScript to ride the popularity of Java at the time—though the two languages are not directly related. Over time, JavaScript evolved from a simple scripting language into a powerful tool capable of building entire web applications.

Today, JavaScript runs **in the browser**, allowing developers to write code that executes on the user's device. All major web browsers have built-in JavaScript engines (like V8 in Chrome and SpiderMonkey in Firefox) that interpret and execute the code. This makes JavaScript a **client-side language**, handling tasks like DOM manipulation, event handling, and animations directly in the browser.

However, JavaScript is not limited to the browser. With the advent of **Node.js**, JavaScript can now run on servers as well. Node.js is a runtime environment that allows JavaScript to be used for server-side scripting, enabling developers to build scalable backend systems, command-line tools, APIs, and more—all using the same language.

1.2 Setting Up Your Environment

Before you start writing JavaScript code, it's important to set up a proper development environment. This section will walk you through three essential tools: a code editor, your web browser's console, and Node.js for running JavaScript outside the browser.

Installing a Code Editor (Visual Studio Code)

A good text editor makes writing and managing code easier. One of the most popular editors for JavaScript is **Visual Studio Code (VS Code)**.

Steps to install VS Code:

1. Go to https://code.visualstudio.com/.
2. Click **Download for Windows/macOS/Linux** depending on your system.
3. Run the installer and follow the instructions.
4. Launch VS Code after installation.

Once open, you can create a new file by clicking **File → New File**, and save it with a `.js` extension (e.g., `script.js`).

> *Tip: Install the "JavaScript (ES6) code snippets" extension from the Extensions tab to speed up your coding.*

Running JavaScript in the Browser Console

All modern browsers (like Chrome, Firefox, Safari, and Edge) come with built-in **developer tools**.

To access the console in **Google Chrome**:

1. Open Chrome.
2. Press `Ctrl + Shift + I` (or `Cmd + Option + I` on Mac).
3. Click on the **Console** tab.
4. Type JavaScript code directly and press `Enter` to execute.

Example:

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

This is a great way to test small snippets of code instantly.

Installing Node.js (for Running JavaScript Outside the Browser)

Node.js lets you run JavaScript from the command line, making it useful for scripting and backend development.

Steps to install Node.js:

1. Go to https://nodejs.org/.
2. Download the **LTS (Long-Term Support)** version for your operating system.
3. Run the installer and follow the setup steps.
4. Open your terminal (Command Prompt or Terminal app).
5. Type `node -v` to confirm the installation.

You can now run `.js` files like this:

```
node script.js
```

With these tools in place—VS Code, browser console, and Node.js—you're ready to write and run JavaScript wherever you need.

1.3 Writing Your First JavaScript Program

Now that your environment is set up, it's time to write and run your very first JavaScript program! In this section, you'll create a simple script that prints "Hello, World!"—the

traditional starting point for learning any new programming language. You'll see how to run it both in a browser and using Node.js.

Running JavaScript in the Browser

1. Open your browser (e.g., Chrome).
2. Right-click anywhere on the page and choose **Inspect**, or press `Ctrl + Shift + I` (`Cmd + Option + I` on Mac).
3. Click on the **Console** tab.

Now type the following code and press **Enter**:

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

You'll see this message printed in the console:

```
Hello, World!
```

The `console.log()` function is used to display messages or outputs in the console. It's one of the most useful tools for debugging and testing your code.

Creating and Running a Script in an HTML File

You can also include JavaScript in a web page. Open your code editor and create a file called `index.html`, then write the following:

```
<!DOCTYPE html>
<html>
<head>
  <title>My First JavaScript Program</title>
</head>
<body>
  <script>
    console.log("Hello from the browser!");
    alert("Hello from the web page!");
  </script>
</body>
</html>
```

Save the file and open it in your browser. You'll see a pop-up saying "Hello from the web page!" and the console will show:

```
Hello from the browser!
```

Running JavaScript with Node.js

1. Open your code editor and create a new file named `hello.js`.
2. Add the following code:

```
console.log("Hello from Node.js!");
```

3. Open a terminal or command prompt.
4. Navigate to the folder where your file is saved.
5. Run the file with this command:

```
node hello.js
```

You should see:

```
Hello from Node.js!
```

By writing your first program in both browser and Node.js environments, you've just taken your first step into JavaScript programming. In the next section, we'll explore how to inspect your code in more detail using developer tools and debuggers.

1.4 Using the Console and Debugger

Modern browsers come equipped with powerful developer tools that allow you to interact with, inspect, and debug your JavaScript code. These tools are essential for diagnosing problems, testing code snippets, and understanding how your scripts behave in real time.

Accessing the Console

To open the Developer Console in most browsers:

- **Chrome**: Press `Ctrl + Shift + I` (or `Cmd + Option + I` on Mac), then click the **Console** tab.
- **Firefox**: Press `F12` or `Ctrl + Shift + K`.
- **Edge**: Press `F12` or `Ctrl + Shift + I`, then go to the **Console** tab.

The console allows you to:

- Run JavaScript commands interactively.
- View outputs from `console.log()`.
- See error messages and stack traces.
- Monitor values and debug code on the fly.

Try entering this in the console:

```
let name = "JavaScript";
console.log("Welcome to", name);
```

You should see:

```
Welcome to JavaScript
```

Basic Debugging with Breakpoints

When your code doesn't behave as expected, debugging helps you step through it line by line to find issues.

Here's how to debug using the **Sources** tab in Chrome:

1. Open your `index.html` file in the browser.
2. Press `Ctrl + Shift + I` to open DevTools.
3. Click the **Sources** tab and find your JavaScript code (either embedded in HTML or from an external `.js` file).
4. Click the line number where you want to pause execution — this sets a **breakpoint**.

Here's a sample script:

```
<script>
  function greet(name) {
    let message = "Hello, " + name + "!";
    console.log(message);
    return message;
  }

  greet("Alice");
</script>
```

Set a breakpoint on the `let message = ...` line. When the page loads and the function runs, execution will pause there.

Inspecting and Stepping Through Code

Once paused:

- Hover over variables to see their current values.
- Use the buttons in the debugger toolbar:
  - **Resume**: Continue execution.
  - **Step Over**: Move to the next line.
  - **Step Into**: Enter into a called function.
  - **Stop**: Cancel the debugging session.

You can also view the **Call Stack**, **Scope**, and **Watch Expressions** to gain deeper insight into what's happening at each stage of execution.

By mastering the console and debugger, you'll be able to test and troubleshoot your code efficiently—an essential skill for every JavaScript developer.

Chapter 2.

Basic Syntax and Statements

1. JavaScript Syntax Rules
2. Comments in JavaScript
3. Variables and Constants (`var`, `let`, `const`)
4. Data Types Overview (String, Number, Boolean, Null, Undefined, Symbol, BigInt)
5. Basic Operators (Arithmetic, Assignment, Comparison, Bitwise)
6. Expressions and Statements
7. Type Conversion and Coercion (Implicit and Explicit)

2 Basic Syntax and Statements

2.1 JavaScript Syntax Rules

Understanding JavaScript's syntax rules is essential to writing code that runs correctly and predictably. JavaScript has a flexible and forgiving syntax, but there are important conventions and rules that every developer should know.

Case Sensitivity

JavaScript is **case-sensitive**, meaning that variable and function names must match exactly in spelling and capitalization.

```
let userName = "Alice";
console.log(username); // NO ReferenceError: username is not defined
```

In the example above, `userName` and `username` are treated as different identifiers.

Statement Termination and Semicolons

In JavaScript, statements are usually ended with a **semicolon (;)**. While JavaScript supports **automatic semicolon insertion (ASI)**, relying on it can lead to confusing bugs.

Correct usage:

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

JavaScript will often insert semicolons for you when they're missing, but this behavior isn't always predictable:

```
// NO Unexpected behavior due to ASI
const getValue = () =>
{
  return
  {
    value: 1
  };
};

console.log(getValue()); // → undefined
```

In the example above, JavaScript inserts a semicolon after `return`, causing it to return `undefined`. To avoid such issues, it's best to **use semicolons consistently**.

Whitespace and Line Breaks

JavaScript ignores extra spaces, tabs, and line breaks, which makes formatting flexible. However, good indentation and spacing improve readability.

```
let x = 10;
let y = 5;
let sum = x + y;
```

The following is also valid but hard to read:

```
let x=10;let y=5;let sum=x+y;
```

Code Blocks

A **code block** is a group of statements wrapped in curly braces `{}`. They're used in functions, conditionals, loops, and other structures.

```
if (true) {
  console.log("This is inside a block.");
}
```

Blocks help define **scope**, especially when using `let` or `const`.

Understanding and following these basic syntax rules will help you avoid common errors and write clean, readable JavaScript code.

2.2 Comments in JavaScript

Comments are lines in your code that are **ignored by the JavaScript engine**. They're used to explain code, add notes, or temporarily disable parts of a script during debugging. Writing clear comments is a best practice that improves code readability and collaboration.

Single-Line Comments

Use `//` for single-line comments. Everything after `//` on that line is ignored.

```
// This is a single-line comment
let age = 30; // This sets the user's age
```

Multi-Line Comments

Use `/* */` to write comments that span multiple lines.

```
/*
  This function calculates
  the sum of two numbers
*/
function add(a, b) {
  return a + b;
}
```

Using Comments for Debugging

You can also comment out lines of code to temporarily disable them:

```
// console.log("This line won't run");
console.log("This line will run");
```

Best Practices

- Use comments to explain complex logic.
- Avoid obvious comments (e.g., `// add 2 + 2` before `2 + 2`).
- Keep comments up-to-date with code changes.

Comments don't affect performance and are crucial for long-term maintenance and understanding of your code.

2.3 Variables and Constants (`var`, `let`, `const`)

In JavaScript, variables are used to store data that can be used and manipulated throughout your program. There are three ways to declare variables: `var`, `let`, and `const`. While they may seem similar at first, they differ in important ways—especially in terms of **scope**, **hoisting**, and **mutability**.

`var` Function Scoped and Hoisted

The `var` keyword was traditionally used to declare variables in JavaScript. It is **function-scoped**, meaning it's only limited to the function in which it's declared, not block-scoped (like inside `if` or `for` blocks). Variables declared with `var` are **hoisted**, meaning they are moved to the top of their scope and initialized as `undefined`.

```
function example() {
  console.log(x); // undefined (hoisted)
  var x = 10;
  console.log(x); // 10
}
example();
```

`let` Block Scoped and Hoisted (But Not Initialized)

Introduced in ES6, `let` is **block-scoped**, which means it is only accessible within the nearest pair of curly braces (`{}`), like in loops or conditional blocks. While `let` is also hoisted, it is not initialized until the line where it's declared, leading to a **temporal dead zone** where accessing it before declaration throws an error.

```
{
  let y = 20;
  console.log(y); // 20
```

```
}
// console.log(y); // NO ReferenceError: y is not defined
```

`const` Block Scoped and Immutable (Binding)

Like `let`, `const` is also **block-scoped** and not initialized during hoisting. The key difference is that variables declared with `const` **cannot be reassigned** after their initial assignment.

```
const pi = 3.14;
// pi = 3.1415; // NO TypeError: Assignment to constant variable
```

However, `const` does **not** make objects or arrays immutable—it only prevents reassignment of the variable itself.

```
const person = { name: "Alice" };
person.name = "Bob"; // YES allowed
```

Best Practices

- Use `**const**` by default for values that won't change.
- Use `**let**` when you expect the value to change (e.g., counters, loop indices).
- **Avoid using `var`** in modern JavaScript—it has confusing scoping and hoisting behavior that often leads to bugs.

2.3.1 Summary

Keyword	Scope	Reassignable	Hoisted	Common Use Case
`var`	Function	Yes	Yes	Legacy code
`let`	Block	Yes	Yes	Changing values
`const`	Block	No	Yes	Fixed bindings and constants

Understanding when and how to use `var`, `let`, and `const` is foundational to writing safe and maintainable JavaScript code.

2.4 Data Types Overview (String, Number, Boolean, Null, Undefined, Symbol, BigInt)

JavaScript has **seven primitive data types**, each representing a single, immutable value. Understanding these types is fundamental to writing reliable code.

String

A `string` represents text and is enclosed in single (`'`), double (`"`), or backtick (`) quotes.

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

Use strings for names, messages, and any text-based content.

Number

The `number` type includes both integers and floating-point values.

```
let age = 30;
let price = 19.99;
console.log(typeof age);   // "number"
console.log(typeof price); // "number"
```

JavaScript does **not** distinguish between integers and floats.

Boolean

Booleans have only two values: `true` and `false`. They're often used in conditions and logic.

```
let isLoggedIn = true;
console.log(typeof isLoggedIn); // "boolean"
```

Null

`null` represents an **intentional absence** of any value. It's often used to reset variables or indicate "no value."

```
let data = null;
console.log(typeof data); // "object" <- known quirk in JavaScript
```

> Note: `typeof null` returns `"object"` due to a legacy bug, but `null` is still a primitive.

Undefined

`undefined` means a variable has been declared but **not assigned a value** yet.

```
let score;
console.log(score);          // undefined
console.log(typeof score);   // "undefined"
```

It also appears when accessing non-existent object properties or missing function returns.

Symbol

Introduced in ES6, `Symbol` is used to create unique identifiers—often for object property keys to avoid naming collisions.

```
let sym = Symbol("id");
console.log(typeof sym); // "symbol"
```

Symbols are not enumerable and are useful in metaprogramming.

BigInt

`BigInt` allows representation of integers larger than the safe limit for regular numbers (`2^53 - 1`).

```
let bigNumber = 12345678901234567890123456789012345678901234567890n;
console.log(typeof bigNumber); // "bigint"
```

BigInt values are created by appending an `n` to the end of the number.

2.4.1 Summary Table

Data Type	Example Value	`typeof` Result
String	`"Hello"`	`"string"`
Number	`42`, `3.14`	`"number"`
Boolean	`true`, `false`	`"boolean"`
Null	`null`	`"object"`
Undefined	`undefined`	`"undefined"`
Symbol	`Symbol("id")`	`"symbol"`
BigInt	`9007199254740991n`	`"bigint"`

Understanding these types helps avoid common bugs and ensures proper handling of values across your programs.

2.5 Basic Operators (Arithmetic, Assignment, Comparison, Bitwise)

Operators in JavaScript are symbols used to perform operations on values and variables. This section covers the most commonly used categories: arithmetic, assignment, comparison, and bitwise operators.

Arithmetic Operators

Arithmetic operators are used to perform basic math operations:

```
let x = 10;
let y = 3;

console.log(x + y); // 13 (addition)
console.log(x - y); // 7  (subtraction)
console.log(x * y); // 30 (multiplication)
console.log(x / y); // 3.333... (division)
console.log(x % y); // 1  (modulo/remainder)
```

These work with both integers and floating-point numbers.

Assignment Operators

Assignment operators assign values to variables. The most basic one is `=`, but there are shorthand forms for arithmetic operations:

```
let a = 5;
a += 2;  // a = a + 2 → 7
a -= 1;  // a = a - 1 → 6
a *= 3;  // a = a * 3 → 18
a /= 2;  // a = a / 2 → 9
a %= 4;  // a = a % 4 → 1

console.log(a); // 1
```

These are useful for compact and readable code.

Comparison Operators

Comparison operators are used to compare two values and return a Boolean (`true` or `false`):

```
console.log(5 > 3);     // true
console.log(5 < 3);     // false
console.log(5 >= 5);    // true
console.log(3 <= 2);    // false
```

== vs. ===

- `==` checks for **equality after type coercion**:

  `js try console.log("5" == 5);  // true`

- `===` checks for **strict equality**, meaning **both value and type must match**:

  `js try console.log("5" === 5); // false`

It's recommended to use `===` and `!==` in most cases to avoid unexpected results due to implicit type conversion.

Bitwise Operators

Bitwise operators work at the binary level. While they are used less often in typical web development, they are important in low-level tasks like graphics, cryptography, or performance

optimization.

```
console.log(5 & 1); // 1 (AND)
console.log(5 | 1); // 5 (OR)
console.log(5 ^ 1); // 4 (XOR)
console.log(~5);    // -6 (NOT)
console.log(5 << 1); // 10 (Left shift)
console.log(5 >> 1); // 2  (Right shift)
```

These operate on 32-bit binary representations of numbers.

2.5.1 Summary

Type	Operator Examples	Description
Arithmetic	+, -, *, /, %	Basic math
Assignment	=, +=, -=, *=, /=, %=	Assign and update values
Comparison	==, ===, !=, !==, <, >, <=, >=	Compare values
Bitwise	&, | , ^, \~, <<, >>\	Binary-level manipulation

Mastering these basic operators lays the foundation for writing logical and expressive JavaScript code.

2.6 Expressions and Statements

Understanding the difference between **expressions** and **statements** is key to writing clear and predictable JavaScript code. These two building blocks define how the language runs code and evaluates results.

What is an Expression?

An **expression** is any piece of code that produces a value. It can be as simple as a number or as complex as a function call. Expressions can be assigned to variables, used in comparisons, or passed to functions.

Examples of expressions:

```
5 + 3           // 8
"Hello" + " World"  // "Hello World"
x = 10          // Assignment expression
```

```
x * 2           // Multiplication (evaluates to 20 if x is 10)
greet("Alice")  // Function call
```

Each of these returns a value when executed.

What is a Statement?

A **statement** is a complete instruction that performs an action. It may contain expressions, but it doesn't necessarily return a value by itself. JavaScript programs are composed of many statements.

Examples of statements:

```
let x = 5;            // Variable declaration (with assignment expression)
if (x > 3) {          // if statement (with condition expression)
  console.log(x);     // function call (another statement)
}
```

Statements **control the flow** and structure of the program, such as conditionals (`if`, `else`), loops (`for`, `while`), and declarations (`let`, `const`).

Combining Expressions and Statements

In real-world code, expressions are often part of statements. For example:

```
let total = price * quantity;
```

Here:

- `price * quantity` is an **expression**.
- The entire line (`let total = ...`) is a **statement**.

Another example:

```
if (score >= 50) {
  console.log("You passed!");
}
```

- `score >= 50` is a **comparison expression**.
- `if (...) { ... }` is a **conditional statement**.
- `console.log(...)` is an **expression statement**.

2.6.1 Summary

Concept	Returns a Value	Example
Expression	YES Yes	`5 + 2`, `"Hi"`, `x * y`
Statement	NO Not directly	`let x = 5;`, `if (...) {}`

Knowing the distinction helps you understand what your code is doing and where each construct fits in the flow of a JavaScript program.

2.7 Type Conversion and Coercion (Implicit and Explicit)

JavaScript is a **loosely typed** language, meaning it often converts values from one type to another automatically. This process is called **type coercion**. Understanding how and when JavaScript changes types is essential to avoid bugs and unexpected results.

Implicit Type Coercion

Implicit coercion happens automatically when JavaScript expects a certain type but receives a different one. It tries to convert values to make the operation work.

Example 1: String concatenation

```
console.log("5" + 3);  // "53"
```

Here, `3` is coerced to a string `"3"` and concatenated with `"5"` resulting in `"53"`.

Example 2: Numeric addition vs subtraction

```
console.log("5" - 2);  // 3
```

The `-` operator forces both operands to be numbers, so `"5"` is converted to `5`, and the subtraction happens normally.

Example 3: Boolean context

```
if ("") {
  console.log("This won't run");
} else {
  console.log("Empty string is falsy");
}
```

An empty string `""` is coerced to `false` in conditions.

Explicit Type Conversion

Explicit conversion is when you manually convert values to a specific type using built-in functions:

- `String(value)` converts to a string.
- `Number(value)` converts to a number.
- `Boolean(value)` converts to a boolean.
- `parseInt(string, radix)` converts a string to an integer with an optional base.

Examples:

```
console.log(String(123));       // "123"
console.log(Number("456"));     // 456
console.log(Boolean(0));        // false
console.log(parseInt("15px"));  // 15
```

Explicit conversion is safer because you control when and how the conversion happens.

Common Pitfalls

- **Loose equality (==) with coercion:**

```
console.log(0 == false);   // true (0 coerced to false)
console.log("0" == 0);     // true ("0" coerced to number 0)
```

- **Unexpected coercion in arithmetic:**

```
console.log("5" * "4");    // 20 (strings coerced to numbers)
console.log("5" + 4);      // "54" (number coerced to string)
```

- **`parseInt` and non-numeric characters:**

```
console.log(parseInt("10abc")); // 10
console.log(parseInt("abc10")); // NaN (Not a Number)
```

- **Boolean conversion quirks:**

Falsy values (converted to `false`) include: `0`, `""`, `null`, `undefined`, `NaN`, and `false`.

2.7.1 Summary

Conversion Type	When It Happens	Examples
Implicit	Automatic, during operations	`"5" + 2` → `"52"`
Explicit	Manually using functions	`Number("5")` → `5`

Understanding and controlling type conversion is essential for writing reliable and predictable JavaScript code. When in doubt, prefer explicit conversion to avoid tricky bugs.

Chapter 3.

Control Flow

1. Conditional Statements: `if`, `else`, `else if`
2. Switch Statement
3. Loops: `for`, `while`, `do...while`, `for...in`, `for...of`
4. Using `break` and `continue`

3 Control Flow

3.1 Conditional Statements: `if`, `else`, `else if`

Conditional statements are the foundation of control flow in JavaScript. They allow your code to make decisions and execute different blocks depending on whether specified conditions evaluate to `true` or `false`. The primary conditional keywords are `if`, `else if`, and `else`.

Basic Syntax and Logic Flow

```
if (condition) {
  // code runs if condition is true
} else if (anotherCondition) {
  // code runs if the first condition is false
  // and anotherCondition is true
} else {
  // code runs if all above conditions are false
}
```

- The `if` statement tests the first condition.
- If `if` is false, JavaScript checks `else if` conditions (if any).
- If none of the conditions are true, the `else` block runs as a fallback.

Simple Example

```
const temperature = 25;

if (temperature > 30) {
  console.log("It's hot outside!");
} else if (temperature >= 15) {
  console.log("The weather is moderate.");
} else {
  console.log("It's cold today.");
}
```

Here, the program checks the temperature and logs different messages depending on the range it falls into.

Compound Conditions with Logical Operators

You can combine multiple conditions using logical operators `&&` (AND) and `||` (OR):

```
const age = 20;
const hasID = true;

if (age >= 18 && hasID) {
  console.log("You can enter the club.");
} else {
  console.log("Entry denied.");
}
```

The message shows only if both conditions are true.

Nested `if` Statements

You can nest `if` statements inside one another to test more complex logic:

```
const score = 85;

if (score >= 60) {
  if (score >= 90) {
    console.log("Grade: A");
  } else if (score >= 80) {
    console.log("Grade: B");
  } else {
    console.log("Grade: C");
  }
} else {
  console.log("Failed the test.");
}
```

This structure first checks if the score passes, then assigns a letter grade based on more specific ranges.

Best Practices for Readability

- **Keep conditions clear and concise.** Avoid overly complex expressions in one line; break them down if necessary.
- **Use braces `{}` even for single-line blocks.** This avoids errors and improves clarity.
- **Prefer `else if` chains over multiple separate `if` statements** when conditions are mutually exclusive.
- **Indent nested blocks consistently.** This visually separates logic levels for easier reading.
- **Use meaningful variable names** to make conditions self-explanatory.

Mastering `if`, `else if`, and `else` statements lets you control the flow of your programs effectively, adapting behavior dynamically based on different inputs or states. As you combine conditions and nest logic, always prioritize clarity to keep your code maintainable and bug-free.

3.2 Switch Statement

The `switch` statement in JavaScript provides a cleaner, more readable way to handle multiple discrete conditions compared to long chains of `if` / `else if` statements. It's especially useful when you want to compare the same variable or expression against various values.

Syntax Overview

```
switch (expression) {
  case value1:
    // code to execute if expression === value1
    break;
  case value2:
    // code to execute if expression === value2
    break;
  // more cases...
  default:
    // code to execute if no cases match
}
```

- The `expression` is evaluated once.
- The value is compared using strict equality (`===`) against each `case`.
- When a matching case is found, its block runs until a `break` is encountered.
- The `break` statement prevents *fall-through*—the execution continuing into the next case.
- The `default` block runs if none of the cases match (optional but recommended).

Practical Example: Day of the Week

```
const day = "Tuesday";

switch (day) {
  case "Monday":
    console.log("Start of the work week.");
    break;
  case "Tuesday":
  case "Wednesday":
  case "Thursday":
    console.log("Midweek days.");
    break;
  case "Friday":
    console.log("Almost weekend!");
    break;
  case "Saturday":
  case "Sunday":
    console.log("Weekend vibes.");
    break;
  default:
    console.log("Not a valid day.");
}
```

Notice how multiple cases can share the same block, as with Tuesday, Wednesday, and Thursday. The code runs the first matching case and stops at the `break`.

Fall-Through Behavior

If you omit `break`, the code continues to the next case:

```
const level = 2;

switch (level) {
  case 1:
    console.log("Level 1");
  case 2:
    console.log("Level 2");
  case 3:
    console.log("Level 3");
    break;
}
```

Output:

```
Level 2
Level 3
```

Since there's no `break` after case 2, it "falls through" and executes case 3 as well. Use fall-through intentionally but carefully to avoid bugs.

Nested Switch Statements

Switch statements can be nested for complex decision trees:

```
const fruit = "apple";
const color = "red";

switch (fruit) {
  case "apple":
    switch (color) {
      case "green":
        console.log("Green apple");
        break;
      case "red":
        console.log("Red apple");
        break;
      default:
        console.log("Unknown apple color");
    }
    break;
  case "banana":
    console.log("Banana selected");
    break;
  default:
    console.log("Unknown fruit");
}
```

Summary

Use `switch` when you compare a single value against many possible matches. It improves readability and organization over lengthy `if/else if` chains. Remember to use `break` to avoid unwanted fall-through, and consider nested switches for layered decisions.

3.3 Loops: `for`, `while`, `do...while`, `for...in`, `for...of`

Loops are essential in JavaScript for repeating actions efficiently. They let you execute a block of code multiple times until a condition changes. This section covers the major loop types, their syntax, use cases, and practical examples.

`for` Loop

The classic `for` loop is ideal when you know how many times you want to iterate, such as looping through arrays by index.

Syntax:

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

Example:

```
const fruits = ["apple", "banana", "cherry"];

for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

This loop starts with `i = 0`, continues while `i` is less than the array length, and increments `i` after each iteration.

`while` Loop

The `while` loop runs as long as its condition remains true. It's useful when you don't know the number of iterations beforehand.

Syntax:

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

Example:

```
let count = 0;

while (count < 5) {
  console.log(count);
  count++;
}
```

`do...while` Loop

This loop guarantees the block runs at least once because the condition is checked **after** the code executes.

Syntax:

```
do {
  // code runs at least once
} while (condition);
```

Example:

```
let number = 0;

do {
  console.log("Number is", number);
  number++;
} while (number < 3);
```

`for...in` Loop

The `for...in` loop iterates over the **enumerable property keys** of an object. It's useful to traverse all keys in an object but should be used carefully with arrays due to potential inherited properties.

Syntax:

```
for (let key in object) {
  // use key to access object[key]
}
```

Example:

```
const person = { name: "Alice", age: 30, city: "Toronto" };

for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

Output:

```
name: Alice
age: 30
city: Toronto
```

`for...of` Loop

Introduced in ES6, `for...of` loops iterate over **iterable objects** like arrays, strings, maps, and sets, returning the values directly.

Syntax:

```
for (let value of iterable) {
  // use value directly
}
```

Example with an array:

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

for (let color of colors) {
  console.log(color);
}
```

When to Use Which Loop?

Loop Type	Use Case
`for`	Fixed iteration count, accessing index explicitly.
`while`	Unknown iteration count, condition checked before run.
`do...while`	Run at least once, then repeat while condition true.
`for...in`	Enumerate keys in objects (properties).
`for...of`	Iterate values of arrays, strings, and other iterables.

Summary

- Use **`for` loops** when counting or iterating with an index.
- Use **`while`** and **`do...while`** for condition-based repetition.
- Use **`for...in`** to loop over object keys (property names).
- Use **`for...of`** to iterate values in arrays and other iterable collections cleanly.

Understanding the strengths of each loop helps you write clearer, more efficient JavaScript code tailored to your data structures and logic needs.

3.4 Using `break` and `continue`

In JavaScript loops and `switch` statements, `break` and `continue` are two powerful control flow tools that let you fine-tune the iteration behavior.

`break`

The `break` statement immediately exits the nearest enclosing loop or `switch` block, skipping any remaining iterations or cases.

Syntax:

```
break;
```

Example: Exiting a loop early

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
```

```
    break; // stop loop when i equals 5
  }
  console.log(i);
}
```

Output:

```
1
2
3
4
```

The loop stops entirely once `i` reaches 5.

Example: Exiting a switch case

```
const color = "green";

switch (color) {
  case "red":
    console.log("Stop");
    break;
  case "green":
    console.log("Go");
    break;  // prevents falling through to default
  default:
    console.log("Caution");
}
```

`continue`

The `continue` statement skips the current iteration of a loop and moves directly to the next one, without executing the remaining code in the loop body.

Syntax:

```
continue;
```

Example: Skipping specific iterations

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue; // skip number 3
  }
  console.log(i);
}
```

Output:

```
1
2
```

```
4
5
```

Using `break` and `continue` in Nested Loops

Both work with nested loops but affect only the innermost loop they are inside.

```
for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 3; j++) {
    if (j === 2) {
      continue; // skips printing when j is 2
    }
    if (i === 3 && j === 1) {
      break; // exits inner loop when i=3 and j=1
    }
    console.log(`i=${i}, j=${j}`);
  }
}
```

Output:

```
i=1, j=1
i=1, j=3
i=2, j=1
i=2, j=3
i=3, j=1
```

Summary

- Use `break` to **exit loops or switches early** when a condition is met.
- Use `continue` to **skip current iterations** but continue looping.
- Both improve efficiency and clarity by controlling flow without complex conditions.

Chapter 4.

Functions

1. Function Declarations and Expressions
2. Parameters and Arguments
3. Return Values
4. Arrow Functions
5. Immediately Invoked Function Expressions (IIFE)
6. Optional Function Chaining in Calls (`fn?.()`)

4 Functions

4.1 Function Declarations and Expressions

Functions are core building blocks in JavaScript, and understanding how they are defined and behave is crucial. Two common ways to define functions are **function declarations** and **function expressions**. They differ in syntax, hoisting, and typical use cases.

Function Declarations

A **function declaration** defines a named function using the `function` keyword and a function name.

Syntax:

```
function greet() {
  console.log("Hello!");
}
```

You can call this function by name anywhere in the scope:

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

Hoisting behavior: Function declarations are *hoisted*, meaning the entire function is loaded into memory during the compile phase. This allows you to call the function **before** its declaration in code:

```
sayHi(); // Works because of hoisting

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

This feature is useful for organizing code with function calls at the top or in different places.

Function Expressions

A **function expression** creates a function inside an expression and can be anonymous or named. It is often assigned to a variable.

Syntax (anonymous function expression):

```
const greet = function() {
  console.log("Hello!");
};
```

Call it via the variable:

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

Named function expression:

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

Here, `sayHello` is local to the function's scope and useful for recursion or debugging.

Hoisting behavior: Function expressions are **not hoisted** like declarations. You cannot call them before the assignment:

```
greet(); // Error: greet is not defined yet

const greet = function() {
  console.log("Hello!");
};
```

When to Use Each?

- **Function declarations** are great for defining core functions upfront, benefiting from hoisting and clear naming.
- **Function expressions** are preferred when functions need to be passed as values, assigned dynamically, or defined conditionally.
- Anonymous function expressions are common in callbacks and event handlers.

Summary Example

```
// Function declaration (hoisted)
hello(); // Works fine

function hello() {
  console.log("Hi from declaration!");
}

// Function expression (not hoisted)
const goodbye = function() {
  console.log("Hi from expression!");
};

goodbye();
```

Understanding these distinctions helps you write cleaner, more predictable code depending on your program's needs.

4.2 Parameters and Arguments

Functions in JavaScript can accept input values through **parameters**, which act as placeholders in the function definition. When calling a function, the actual values you provide are

called **arguments**.

Parameters vs. Arguments

```
function greet(name) {    // 'name' is a parameter
  console.log("Hello, " + name);
}

greet("Alice");           // "Alice" is the argument
```

Here, `name` is the parameter, and `"Alice"` is the argument passed when the function is called.

Handling Missing or Excess Arguments

JavaScript functions are flexible — they don't require the exact number of arguments as parameters:

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

console.log(multiply(2, 3)); // 6
console.log(multiply(2));    // NaN because b is undefined
console.log(multiply(2, 3, 4)); // 6, extra argument ignored
```

If arguments are missing, the corresponding parameters are `undefined`. Extra arguments beyond parameters are ignored unless handled explicitly.

Default Parameters

You can assign default values to parameters, so missing arguments fall back to those defaults:

```
function greet(name = "Guest") {
  console.log("Hello, " + name);
}

greet("Bob");    // Hello, Bob
greet();         // Hello, Guest
```

Default parameters ensure your function behaves predictably even if arguments are omitted.

Rest Parameters (Brief Intro)

To handle an indefinite number of arguments, you can use the **rest parameter** syntax `...`:

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3));   // 6
console.log(sum(4, 5));      // 9
console.log(sum());          // 0
```

`...numbers` collects all passed arguments into an array called `numbers`, letting you work with variable inputs easily.

Summary

- **Parameters** define what inputs a function expects.
- **Arguments** are actual values passed during calls.
- Missing arguments default to `undefined` unless defaults are set.
- Extra arguments are ignored unless handled via rest parameters.
- Use **default parameters** to provide fallback values.
- Use **rest parameters** (`...`) to accept any number of arguments gracefully.

This flexibility makes JavaScript functions powerful and adaptable to various calling scenarios.

4.3 Return Values

Functions often perform calculations or operations and then **return** a value to the caller using the `return` statement. The returned value can be any valid JavaScript data type, including numbers, strings, objects, or even other functions.

Using the `return` Statement

The syntax is simple:

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

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

When `return` executes, the function immediately exits and sends the specified value back. If a function doesn't have a `return` statement or it runs without hitting one, it implicitly returns `undefined`.

```
function greet(name) {
  console.log("Hello, " + name);
}

const result = greet("Alice");
console.log(result); // undefined
```

Returning Different Data Types

Functions can return any type of value:

```
// Returning a string
function getName() {
  return "Charlie";
}

// Returning an object
function createUser() {
  return { name: "Dana", age: 28 };
}

// Returning another function (closure)
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

console.log(getName()); // "Charlie"
console.log(JSON.stringify(createUser(),null,2)); // { name: "Dana", age: 28 }

const double = multiplier(2);
console.log(double(5)); // 10
```

Practical Usage

- Use the returned value for further processing:

```
const area = calculateArea(5, 10);
console.log("Area:", area);
```

- You can also call functions without using their return value (e.g., when the function performs side effects):

```
logMessage("Hello, world!"); // Function logs but returns undefined
```

Summary

- The `return` statement sends a value back from a function to its caller.
- Functions without an explicit `return` return `undefined` by default.
- Return values can be primitives, objects, or even other functions.
- Capturing returned values allows chaining operations and building flexible code.

4.4 Arrow Functions

Arrow functions, introduced in ES6, provide a concise way to write functions in JavaScript. They differ from traditional functions in syntax, return behavior, and the way they handle the `this` keyword.

Basic Syntax

An arrow function uses the `=>` syntax:

```
const add = (a, b) => {
  return a + b;
};
console.log(add(2, 3)); // 5
```

Here, the parameters `(a, b)` are followed by an arrow and a function body.

Implicit Return

If the function body contains a single expression, you can omit the braces `{}` and the `return` keyword — the expression is implicitly returned:

```
const multiply = (x, y) => x * y;
console.log(multiply(4, 5)); // 20
```

For a single parameter, parentheses can also be omitted:

```
const square = x => x * x;
console.log(square(6)); // 36
```

If there are no parameters, use empty parentheses:

```
const greet = () => "Hello!";
console.log(greet()); // Hello!
```

Lexical `this` Binding

Unlike traditional functions, arrow functions **do not have their own `this`** context. Instead, they inherit `this` from the surrounding scope (lexical `this`). This makes arrow functions especially useful for callbacks or methods where you want to maintain the outer `this`.

Example:

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++;
    console.log(this.age);
  }, 1000);
}

const p = new Person();
// The arrow function inside setInterval keeps the 'this' bound to the Person instance
```

With a traditional function, you'd need to bind `this` or use `self = this` trick, but arrow functions simplify this.

Limitations

- Arrow functions **cannot** be used as constructors (i.e., with `new`).
- They do not have their own `arguments` object.
- They are less suitable for object methods when you want the method's own `this`.

More Complex Example

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(n => n * 2);
console.log(doubled); // [2, 4, 6, 8]
```

Summary

- Arrow functions offer concise syntax, especially for short functions.
- Implicit returns reduce boilerplate for simple expressions.
- Lexical `this` binding solves common context issues.
- They are ideal for callbacks and functional programming but not for methods needing their own `this` or constructors.

Mastering arrow functions helps write cleaner, more readable modern JavaScript code.

4.5 Immediately Invoked Function Expressions (IIFE)

An **Immediately Invoked Function Expression (IIFE)** is a function that is defined and executed instantly. It creates a new scope, which helps avoid polluting the global namespace and keeps variables private.

Basic Syntax

The classic IIFE syntax wraps a function expression in parentheses, followed by `()` to invoke it immediately:

```
(function() {
  console.log("This runs immediately!");
})();
```

This defines an anonymous function and calls it right away.

Why Use IIFE?

- **Variable privacy:** Variables declared inside an IIFE are scoped locally and don't leak into the global scope.
- **Module pattern:** IIFEs can encapsulate related code, exposing only selected parts via returned objects.
- **Avoid polluting global scope:** Especially useful before ES6 modules were standard-

ized.

Practical Example: Variable Privacy

```
(function() {
  const secret = "hidden";
  console.log("Inside IIFE:", secret);
})();

console.log(typeof secret); // undefined - secret is not global
```

The variable `secret` is inaccessible outside the IIFE, preventing accidental access or conflicts.

IIFE Returning an Object (Module Pattern)

```
const counter = (function() {
  let count = 0;

  return {
    increment() {
      count++;
      console.log(count);
    },
    reset() {
      count = 0;
      console.log("Reset");
    }
  };
})();

counter.increment(); // 1
counter.increment(); // 2
counter.reset();     // Reset
```

The internal `count` variable stays private, while public methods control it.

IIFE with Arrow Functions

You can also write IIFEs using arrow functions, but parentheses around the function expression are still required:

```
(() => {
  console.log("Arrow function IIFE");
})();
```

Summary

IIFEs are a powerful way to:

- Run code immediately with isolated scope.
- Protect variables from global pollution.
- Implement simple module patterns in JavaScript.

While ES6 modules have largely replaced IIFEs for modularity, they remain useful for quick, self-contained scripts and legacy code.

4.6 Optional Function Chaining in Calls (`fn?.()`)

The **optional chaining operator** (`?.`) lets you safely call functions that might be `undefined` or `null` without causing runtime errors. This is especially useful in complex code where a function may or may not exist at a certain point.

Basic Usage

Normally, calling a function that is `undefined` throws an error:

```
let fn;

fn(); // TypeError: fn is not a function
```

With optional chaining, you can safely attempt to call `fn` only if it exists:

```
fn?.(); // Does nothing and does NOT throw an error
```

Here, `fn?.()` checks if `fn` is not `null` or `undefined` before invoking it.

Example: Comparing Normal vs Optional Call

```
const user = {
  greet() {
    console.log("Hello!");
  }
};

user.greet();  // Output: Hello!

const maybeGreet = user.maybeGreet;

maybeGreet();  // Error: maybeGreet is not a function

maybeGreet?.(); // No error, nothing happens
```

Typical Use Cases

- **Calling optional callback functions:** When you pass callbacks that may or may not be provided.

```
function process(data, callback) {
  // call callback only if it exists
  callback?.(data);
}
```

```
process("data");          // Safe, no callback provided
process("data", console.log); // Calls console.log with data
```

- **Working with uncertain object structures:** When you're not sure if a method exists deep inside an object.

```
const obj = {
  method: null
};

obj.method?.(); // Safe, no error thrown
```

Summary

The `?.()` syntax simplifies safe function invocation by avoiding explicit existence checks. It helps write cleaner code, especially when dealing with optional or dynamically assigned functions.

Chapter 5.

Objects and Arrays

1. Object Literals and Properties
2. Accessing and Modifying Object Properties
3. Computed Property Names (`[expr]: value`)
4. Arrays and Array Literals
5. Array Methods (push, pop, shift, unshift, slice, splice)
6. Iterating Over Arrays and Objects (`for...in`, `for...of`)

5 Objects and Arrays

5.1 Object Literals and Properties

In JavaScript, **object literals** provide a simple way to create objects — collections of key-value pairs — using a clean and readable syntax.

Basic Syntax

An object literal is defined with curly braces `{}`, containing properties as key-value pairs:

```
const person = {
  name: "Alice",
  age: 30,
  isStudent: false
};
```

Here, `name`, `age`, and `isStudent` are **property keys**, while `"Alice"`, `30`, and `false` are their corresponding **values**.

Property Keys

Property keys in JavaScript objects can be:

- **Strings** (the most common): Keys are typically written as unquoted identifiers, but can also be quoted strings.

```
const obj = {
  firstName: "John",
  "last-name": "Doe"  // Quoted key because of the hyphen
};
```

- **Numbers:** Numeric keys are allowed and are treated as strings internally.

```
const scores = {
  1: 95,
  2: 87
};
```

- **Symbols:** A less common but unique type used as keys, useful for defining properties that won't clash.

```
const sym = Symbol("id");
const obj = {
  [sym]: 123
};
```

Shorthand Property Notation

When variable names match property names, you can use **shorthand notation** to create properties:

```
const name = "Bob";
const age = 25;

const user = { name, age };
console.log(JSON.stringify(user,null,2)); // { name: "Bob", age: 25 }
```

This is equivalent to writing `{ name: name, age: age }` but more concise.

Example: Defining an Object with Multiple Properties

```
const book = {
  title: "JavaScript Basics",
  author: "Jane Smith",
  year: 2024,
  isPublished: true
};
```

You can easily add, modify, or delete properties later as well, but object literals offer a straightforward way to initialize objects with meaningful data.

Understanding object literals and property types is fundamental for working with JavaScript's flexible and powerful object system.

5.2 Accessing and Modifying Object Properties

In JavaScript, you can access and modify object properties using two main approaches: **dot notation** and **bracket notation**. Both allow reading, updating, adding, or deleting properties, but each has its use cases.

Dot Notation

Dot notation is the most straightforward way to access or set a property when you know the exact property name and it follows identifier naming rules.

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

console.log(person.name);  // Output: Alice
person.age = 31;           // Update age
console.log(person.age);   // Output: 31
```

Bracket Notation

Bracket notation is more flexible. It uses a string or expression inside square brackets `[]`, allowing you to access properties dynamically or those with special characters or reserved words.

```
const car = {
  "model-name": "Sedan",
  year: 2020,
  "for": "sale"
};

console.log(car["model-name"]); // Output: Sedan
console.log(car["year"]);       // Output: 2020
console.log(car["for"]);        // Output: sale
```

Using Variables for Property Names

Bracket notation is required when property keys are stored in variables or are computed dynamically:

```
const key = "name";
const user = { name: "Bob" };

console.log(user[key]); // Output: Bob
```

Dot notation won't work here because `user.key` looks for the literal key `"key"`.

Adding and Deleting Properties

Adding new properties is as simple as assigning a value:

```
const obj = {};
obj.newProp = 123;         // Using dot notation
obj["anotherProp"] = 456; // Using bracket notation
console.log(JSON.stringify(obj,null,2));
```

To delete a property, use the `delete` keyword:

```
delete obj.newProp;
console.log(obj); // { anotherProp: 456 }
```

Summary

- Use **dot notation** for fixed, valid identifiers.
- Use **bracket notation** for property names with spaces, special characters, reserved words, or dynamic keys.
- Both notations support reading, updating, adding, and deleting properties.
- Bracket notation is essential for computed or variable-based property access.

These flexible options make working with object properties both powerful and adaptable.

5.3 Computed Property Names (`[expr]: value`)

Computed property names allow you to dynamically define property keys in an object literal using expressions. Instead of fixed property names, you can use any valid JavaScript expression inside square brackets `[ ]`, and its evaluated result becomes the property key.

Syntax

```
const obj = {
  [expression]: value
};
```

The expression inside the brackets is evaluated, and the result is used as the property key.

Basic Example

```
const propName = "score";

const player = {
  [propName]: 100
};

console.log(player.score); // 100
```

Here, the value of `propName` ("score") is used as the property key.

Using Expressions for Keys

You can also use more complex expressions:

```
const prefix = "user_";
const id = 42;

const data = {
  [prefix + id]: "John Doe"
};

console.log(data.user_42); // John Doe
```

This lets you build property keys dynamically based on variables or calculations.

Useful Scenarios

- Creating objects with dynamic keys based on input data.
- Defining properties using constants or computed strings.
- Implementing flexible data structures like maps without using Map objects.
- Writing concise code without separately assigning properties after object creation.

Example: Dynamic Event Handlers

```
const eventType = "click";

const handlers = {
  [eventType]: () => console.log("Clicked!"),
  mouseover: () => console.log("Mouse over!")
};

handlers.click();      // Clicked!
handlers.mouseover();  // Mouse over!
```

Summary

Computed property names offer a clean and expressive way to define object properties when keys depend on variables or expressions. This feature enhances flexibility and reduces repetitive code when dealing with dynamic data structures.

5.4 Arrays and Array Literals

Arrays in JavaScript are special objects designed to store ordered collections of values. Unlike plain objects, arrays use numeric indices to access elements and maintain the order of insertion.

Declaring Arrays with Literals

The most common way to create an array is with **array literals** using square brackets `[]`:

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[0]); // apple
```

Here, `"apple"` is at index `0`, `"banana"` at `1`, and so on.

Arrays vs. Objects

While arrays are technically objects, they are optimized to work with sequential numeric keys (indices), which makes them ideal for lists. Objects use arbitrary keys (strings or symbols), but arrays use integers starting at zero.

Arrays with Mixed Types

JavaScript arrays can hold elements of **different types**:

```
const mixedArray = [42, "hello", true, { name: "Alice" }, [1, 2, 3]];
console.log(mixedArray[3].name); // Alice
```

This flexibility lets you combine values of any type in one array.

Empty Slots and Length Property

Arrays can have **empty slots** (holes), which are positions without assigned values:

```
const arr = [1, , 3];
console.log(arr.length); // 3
console.log(arr[1]);     // undefined
```

The `length` property always reflects the highest index plus one, regardless of empty slots.

Summary

- Arrays store ordered collections accessible by numeric indices.
- Created using square brackets `[]`.
- Can contain mixed data types.
- Support empty slots, but those slots return `undefined`.
- The `length` property shows the total number of elements, including empty slots.

Arrays form a fundamental part of JavaScript data structures, providing a versatile way to work with sequences of data.

5.5 Array Methods (push, pop, shift, unshift, slice, splice)

JavaScript arrays come with several built-in methods to modify or extract elements. Understanding these methods helps you manipulate arrays efficiently. Below are some of the most common methods:

`push()`

Adds one or more elements to the **end** of an array and returns the new length.

```
const arr = [1, 2, 3];
console.log("Before push:", arr);

arr.push(4);
console.log("After push:", arr); // [1, 2, 3, 4]
```

`pop()`

Removes the **last** element from the array and returns it. It **mutates** the array.

```
const arr = [1, 2, 3];
console.log("Before pop:", arr);

const removed = arr.pop();
console.log("Removed:", removed);   // 3
console.log("After pop:", arr);     // [1, 2]
```

`shift()`

Removes the **first** element from the array and returns it, shifting all other elements left.

```
const arr = [1, 2, 3];
console.log("Before shift:", arr);

const removed = arr.shift();
console.log("Removed:", removed);  // 1
console.log("After shift:", arr);  // [2, 3]
```

`unshift()`

Adds one or more elements to the **beginning** of an array and returns the new length.

```
const arr = [2, 3];
console.log("Before unshift:", arr);

arr.unshift(1);
console.log("After unshift:", arr); // [1, 2, 3]
```

`slice()`

Creates a **shallow copy** of a portion of an array **without mutating** the original. It takes a start index (inclusive) and an optional end index (exclusive).

```
const arr = [1, 2, 3, 4, 5];
const sliced = arr.slice(1, 4);

console.log("Original array:", arr);   // [1, 2, 3, 4, 5]
console.log("Sliced array:", sliced);  // [2, 3, 4]
```

`splice()`

Changes the content of an array by **removing, replacing, or adding** elements at a specified index. It **mutates** the original array.

Syntax:

```
array.splice(start, deleteCount, item1, item2, ...)
```

- `start`: Index to start changing the array.
- `deleteCount`: Number of elements to remove.
- Additional arguments: Items to add at `start`.

```
const arr = [1, 2, 3, 4, 5];
console.log("Before splice:", arr);

// Remove 2 elements starting at index 1, then add 9 and 10
const removed = arr.splice(1, 2, 9, 10);

console.log("Removed elements:", removed); // [2, 3]
```

```
console.log("After splice:", arr);          // [1, 9, 10, 4, 5]
```

5.5.1 Summary

Method	Purpose	Mutates Original?	Returns
`push`	Add elements to end	Yes	New length
`pop`	Remove last element	Yes	Removed element
`shift`	Remove first element	Yes	Removed element
`unshift`	Add elements to start	Yes	New length
`slice`	Extract portion without change	No	New array
`splice`	Remove/add/replace elements	Yes	Array of removed elements

These methods offer flexible ways to work with arrays—whether adding, removing, or copying elements—making them essential tools in JavaScript programming.

5.6 Iterating Over Arrays and Objects (`for...in`, `for...of`)

JavaScript offers two powerful loop constructs to iterate over collections: `for...in` and `for...of`. Though similar in syntax, they serve different purposes and work differently depending on whether you are iterating over objects or arrays.

`for...in` Iterating Over Object Keys

The `for...in` loop iterates over the **enumerable property keys** of an object. It works well for objects but should be used cautiously with arrays.

```
const person = {
  name: "Alice",
  age: 30,
  city: "Toronto"
};

for (const key in person) {
  console.log(key, ":", person[key]);
}
// Output:
// name : Alice
// age : 30
// city : Toronto
```

Use case: Access all property names in an object, including those added dynamically.

Caution with arrays: Because `for...in` iterates over all enumerable keys (including inherited ones), it is not recommended for arrays.

`for...of` Iterating Over Values

The `for...of` loop iterates over **iterable objects**, such as arrays, strings, Maps, and Sets, providing direct access to the **values**:

```
const fruits = ["apple", "banana", "cherry"];

for (const fruit of fruits) {
  console.log(fruit);
}
// Output:
// apple
// banana
// cherry
```

Use case: Loop through array elements or other iterable values.

Common Pitfalls

- Using `for...in` on arrays may yield unexpected results, such as iterating over inherited properties or non-index keys:

```
Array.prototype.customMethod = function() {};
const arr = [10, 20, 30];

for (const index in arr) {
  console.log(index);  // Might print "0", "1", "2", and "customMethod"
}
```

- `for...of` works only on iterable objects (arrays, strings, etc.), so it cannot be used directly on plain objects:

```
const obj = {a: 1, b: 2};
// for (const val of obj) {} // TypeError: obj is not iterable
```

Iterating Over Object Values

To iterate over object values, use `Object.values()` with `for...of`:

```
const obj = {a: 1, b: 2, c: 3};

for (const value of Object.values(obj)) {
  console.log(value);
}
// Output:
// 1
// 2
// 3
```

5.6.1 Summary

Loop	Iterates Over	Use Case
`for...in`	Object keys (property names)	Objects, but **avoid for arrays**
`for...of`	Iterable values	Arrays, strings, and iterables

Choosing the right loop improves code clarity and avoids common bugs when iterating over arrays or objects.

Chapter 6.

Advanced Functions

1. Function Scope and Closures
2. Default Parameters
3. Rest Parameters and Spread Syntax
4. Callback Functions
5. Higher-Order Functions (map, filter, reduce)

6 Advanced Functions

6.1 Function Scope and Closures

Understanding **function scope** and **closures** is key to mastering JavaScript's behavior with variables and functions.

Function Scope and Lexical Scope

In JavaScript, **function scope** means variables declared inside a function are only accessible within that function and its nested functions. This prevents variable collisions and keeps data encapsulated.

Lexical scope refers to how JavaScript resolves variable names based on their location in the source code. Inner functions can access variables declared in their outer (parent) functions.

```
function outer() {
  const greeting = "Hello";

  function inner() {
    console.log(greeting);  // Accesses outer function variable
  }

  inner();
}

outer();  // Output: Hello
```

Here, the inner function can "see" the variable `greeting` because it's lexically inside `outer`.

What Are Closures?

A **closure** occurs when an inner function retains access to its outer function's variables even after the outer function has finished executing.

```
function makeCounter() {
  let count = 0;

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

const counter = makeCounter();

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

Even though `makeCounter()` has completed, the returned function still **remembers** the `count` variable and can modify it. This behavior is a closure.

Practical Uses of Closures

1. Data Privacy

Closures allow you to create private variables inaccessible from outside:

```
function secretHolder(secret) {
  return {
    getSecret() {
      return secret;
    },
    setSecret(newSecret) {
      secret = newSecret;
    }
  };
}

const mySecret = secretHolder("my password");
console.log(mySecret.getSecret()); // "my password"
mySecret.setSecret("new password");
console.log(mySecret.getSecret()); // "new password"
```

The `secret` variable cannot be accessed directly; only through provided methods.

2. Function Factories

You can generate specialized functions with preset variables:

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
const triple = multiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Each returned function "remembers" its own `factor` value.

6.1.1 Summary

- **Function scope** limits variables to their function and nested scopes.
- **Lexical scope** means inner functions access outer variables based on their code position.
- **Closures** allow inner functions to retain access to outer variables even after the outer function finishes.
- Closures enable **data privacy** and **function factories**, making JavaScript functions more powerful and flexible.

Mastering closures is essential for writing advanced, clean, and modular JavaScript code.

6.2 Default Parameters

JavaScript allows you to assign **default values** to function parameters. These defaults are used when the corresponding argument is either missing or explicitly `undefined`.

Basic Syntax

You define defaults directly in the function signature using the `=` operator:

```
function greet(name = "Guest") {
  console.log("Hello, " + name + "!");
}

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

Here, if no argument is passed, `name` defaults to `"Guest"`.

Default Parameters with Expressions

Default values can depend on earlier parameters:

```
function calculateTotal(price, tax = price * 0.1) {
  return price + tax;
}

console.log(calculateTotal(100));       // 110 (tax defaults to 10)
console.log(calculateTotal(100, 20));   // 120 (tax overridden)
```

In this example, the `tax` parameter defaults to 10% of `price` if no tax argument is provided.

Behavior with `undefined`

Passing `undefined` triggers the default value, but other falsy values like `null` or `0` do **not**:

```
function showMessage(message = "No message") {
  console.log(message);
}

showMessage(undefined);  // Output: No message
showMessage(null);       // Output: null
showMessage(0);          // Output: 0
```

6.2.1 Summary

- Default parameters provide a clean way to handle missing arguments.
- Defaults are set in the function signature using `=`.
- Later parameters can depend on earlier ones.
- Only `undefined` triggers the default, not other falsy values.

Default parameters help write safer and more concise functions by avoiding manual argument checks inside the function body.

6.3 Rest Parameters and Spread Syntax

JavaScript provides two related but distinct features—**rest parameters** and **spread syntax**—that help work with collections of values flexibly.

Rest Parameters

Rest parameters allow a function to accept **an indefinite number of arguments** as an array. You use the `...` syntax before the last parameter in a function definition:

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3));       // 6
console.log(sum(5, 10, 15, 20)); // 50
```

Here, `numbers` collects all arguments into an array, no matter how many are passed.

Spread Syntax

Spread syntax also uses `...`, but it **expands** an array or iterable into individual elements. This is useful for function calls, array literals, or object literals.

Example: Expanding an array into arguments

```
const nums = [4, 5, 6];
console.log(Math.max(...nums)); // 6
```

Without spread, `Math.max(nums)` would not work as expected because `Math.max` expects separate numeric arguments.

Spread in Array Literals

You can merge or copy arrays easily with spread:

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

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

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

Spread in Object Literals

Since ES2018, spread works with objects to copy or merge properties:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };

const merged = { ...obj1, ...obj2 };

console.log(JSON.stringify(merged,null,2)); // { a: 1, b: 3, c: 4 }
```

Note: Properties in the later objects overwrite earlier ones.

Combined Example: Using Rest and Spread Together

```
function joinStrings(separator, ...strings) {
  return strings.join(separator);
}

const words = ["hello", "world", "from", "JS"];
console.log(joinStrings(" ", ...words)); // "hello world from JS"
```

- The function accepts a separator plus any number of strings (`rest`).
- The array `words` is expanded into individual arguments (`spread`) when calling.

6.3.1 Summary

- **Rest parameters** (`...args`) collect multiple function arguments into an array.
- **Spread syntax** (`...arrayOrObject`) expands arrays or objects into individual elements or properties.
- Both improve flexibility when handling variable numbers of arguments or merging data structures.

Together, rest and spread make JavaScript functions and data manipulation more expressive and concise.

6.4 Callback Functions

A **callback function** is a function passed as an argument to another function, intended to be called later—often after some operation completes. Callbacks are fundamental to JavaScript's

asynchronous programming and event handling.

Why Use Callbacks?

Callbacks let you control the order of execution, especially when dealing with operations that take time, such as reading files, making network requests, or waiting for user actions.

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 and invoked inside `greet` after the greeting.

Callbacks in Asynchronous Programming

Callbacks often handle tasks that complete asynchronously:

```
setTimeout(() => {
  console.log("Executed after 2 seconds");
}, 2000);
```

The arrow function is a callback executed after a 2-second delay.

Event Handling Example

Callbacks are common in event listeners:

```
button.addEventListener("click", () => {
  console.log("Button clicked!");
});
```

The function passed to `addEventListener` is called when the event happens.

Common Patterns and Pitfalls

- **Avoid callback hell:** Nested callbacks can make code hard to read and maintain.

```
doFirst(() => {
  doSecond(() => {
    doThird(() => {
      // Deep nesting!
```

```
    });
  });
});
```

- **Error-first callbacks:** In Node.js and other libraries, callbacks often receive an error as the first argument to handle exceptions gracefully:

```
fs.readFile("file.txt", (err, data) => {
  if (err) {
    console.error("Error:", err);
    return;
  }
  console.log(data);
});
```

6.4.1 Summary

Callbacks are functions passed as arguments and executed later, enabling asynchronous workflows and event-driven programming. Understanding callbacks helps write responsive and modular JavaScript applications.

6.5 Higher-Order Functions (map, filter, reduce)

Higher-order functions are functions that **take other functions as arguments** or **return functions**. In JavaScript, array methods like `map`, `filter`, and `reduce` are common higher-order functions used to process and transform data in a clean, declarative way.

`map()` Transforming Arrays

The `map()` method creates a **new array** by applying a function to every element of an existing array. It's perfect for transforming data.

Syntax:

```
const newArray = array.map(callback);
```

Example: Convert an array of temperatures in Celsius to Fahrenheit:

```
const celsius = [0, 20, 30, 100];
const fahrenheit = celsius.map(temp => (temp * 9/5) + 32);

console.log(fahrenheit); // [32, 68, 86, 212]
```

`filter()` Selecting Elements

The `filter()` method creates a **new array** containing only elements that satisfy a condition specified by the callback. It's great for extracting subsets.

Syntax:

```
const filteredArray = array.filter(callback);
```

Example: Get only adults from an array of ages:

```
const ages = [12, 18, 21, 16, 30];
const adults = ages.filter(age => age >= 18);

console.log(adults); // [18, 21, 30]
```

`reduce()` Reducing to a Single Value

The `reduce()` method applies a function to accumulate array elements into a **single value**. It can be used for sums, averages, or more complex reductions.

Syntax:

```
const result = array.reduce((accumulator, currentValue) => {
  // Combine accumulator and currentValue
}, initialValue);
```

Example: Calculate the sum of numbers:

```
const numbers = [10, 20, 30, 40];
const sum = numbers.reduce((total, num) => total + num, 0);

console.log(sum); // 100
```

Real-World Example: Processing Orders

Imagine an array of orders with prices and quantities:

```
const orders = [
  { price: 50, quantity: 2 },
  { price: 30, quantity: 1 },
  { price: 20, quantity: 4 }
];

// Calculate total cost
const totalCost = orders.reduce((total, order) => total + order.price * order.quantity, 0);

console.log(totalCost); // 190
```

6.5.1 Summary

Method	Purpose	Returns
`map`	Transform every element	New array
`filter`	Select elements by condition	New array
`reduce`	Aggregate elements	Single value (any type)

By mastering these higher-order functions, you can write clearer, more expressive code that processes data without explicit loops or temporary variables.

Chapter 7.

The `this` Keyword

1. Understanding `this` in Different Contexts
2. `call`, `apply`, and `bind` Methods

7 The `this` Keyword

7.1 Understanding `this` in Different Contexts

The `this` keyword in JavaScript is a fundamental but often confusing concept. It refers to the object that is currently executing the function, but its exact value depends on **how** and **where** the function is called.

Default Binding (Global Context)

When a function is called in the **global context** (not as a method), `this` refers to the global object (`window` in browsers, `global` in Node.js). In strict mode (`'use strict'`), `this` is `undefined`.

```
function show() {
  console.log(this);
}

show(); // In browsers: Window object (or undefined in strict mode)
```

Implicit Binding (Object Method)

When a function is called as a **method of an object**, `this` refers to the **object before the dot**.

```
const person = {
  name: "Alice",
  greet() {
    console.log("Hello, " + this.name);
  }
};

person.greet(); // Hello, Alice
```

Here, `this` points to `person`.

Explicit Binding (`call`, `apply`, `bind`)

You can explicitly set `this` using `call()`, `apply()`, or `bind()`:

```
function greet() {
  console.log("Hello, " + this.name);
}

const user = { name: "Bob" };

greet.call(user);  // Hello, Bob
greet.apply(user); // Hello, Bob

const boundGreet = greet.bind(user);
boundGreet();      // Hello, Bob
```

- `call` and `apply` invoke the function immediately with `this` set to the first argument.
- `bind` returns a new function permanently bound to the specified `this`.

New Binding (Constructor Functions)

When a function is called with the `new` keyword, `this` points to the **newly created object**.

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

const p = new Person("Charlie");
console.log(p.name); // Charlie
```

Arrow Functions and Lexical `this`

Arrow functions **do not have their own `this`**. Instead, `this` is lexically inherited from the enclosing scope at the time the arrow function is defined.

```
const obj = {
  name: "Dana",
  regularFunc() {
    console.log("regularFunc this:", this.name);
  },
  arrowFunc: () => {
    console.log("arrowFunc this:", this.name);
  }
};

obj.regularFunc(); // regularFunc this: Dana
obj.arrowFunc();   // arrowFunc this: undefined (or window.name if exists)
```

In this case, `arrowFunc`'s `this` is not `obj` but the outer scope (often the global object).

Common Pitfalls

- Losing `this` context when passing object methods as callbacks:

```
const user = {
  name: "Eve",
  greet() {
    console.log(this.name);
  }
};

setTimeout(user.greet, 1000); // undefined or window.name, because `this` is lost
```

To fix it, use `bind` or arrow functions:

```
setTimeout(user.greet.bind(user), 1000);
```

7.1.1 Summary of Binding Rules

Binding Type	When It Happens	What `this` Refers To
Default	Plain function call	Global object (or `undefined` in strict mode)
Implicit	Called as object method	Object owning the method
Explicit	Using `call`, `apply`, `bind`	Object explicitly specified
New	Function called with `new`	Newly created object
Lexical (Arrow)	Arrow functions	`this` from surrounding scope

Understanding `this` requires knowing **how a function is called**. With practice, you'll master this crucial part of JavaScript's behavior.

7.2 `call`, `apply`, and `bind` Methods

The methods `call()`, `apply()`, and `bind()` are powerful tools in JavaScript that let you **explicitly set the value of `this`** when invoking a function. They enable borrowing methods from other objects, controlling execution context, and even partial function application.

`call()` Immediate Invocation with Arguments

`call` invokes a function immediately, setting `this` to the first argument, followed by individual arguments.

```
function greet(greeting, punctuation) {
  console.log(greeting + ", " + this.name + punctuation);
}

const user = { name: "Alice" };

greet.call(user, "Hello", "!"); // Hello, Alice!
```

`apply()` Immediate Invocation with Argument Array

`apply` is similar to `call`, but arguments are passed as an **array** or array-like object.

```
greet.apply(user, ["Hi", "!!!"]); // Hi, Alice!!!
```

This is handy when the arguments are already in an array.

`bind()` Returns a New Function with Bound `this`

Unlike `call` and `apply`, `bind` does **not** call the function immediately. Instead, it returns a **new function** with `this` permanently bound to the specified object.

```
const greetUser = greet.bind(user);

greetUser("Hey", "?"); // Hey, Alice?
```

This is useful for delayed execution or passing callbacks where you want to preserve `this`.

Borrowing Methods with `call` or `apply`

You can use `call` or `apply` to borrow methods from one object and use them with another:

```
const person1 = { name: "Bob" };
const person2 = { name: "Carol" };

function sayName() {
  console.log(this.name);
}

sayName.call(person1); // Bob
sayName.call(person2); // Carol
```

Partial Application with `bind`

`bind` can also fix some arguments upfront (known as **partial application**):

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

const double = multiply.bind(null, 2);

console.log(double(5)); // 10
```

Here, `double` is a new function where `a` is always `2`.

7.2.1 Summary

Method	Purpose	Syntax Example	Key Difference
`call`	Invoke immediately with args	`fn.call(thisArg, arg1, arg2, ...)`	Arguments passed individually
`apply`	Invoke immediately with arg array	`fn.apply(thisArg, [arg1, arg2])`	Arguments passed as an array
`bind`	Returns new bound function	`const f = fn.bind(thisArg, arg1)`	Does **not** invoke immediately

Mastering these methods allows precise control over function context and versatile function usage in JavaScript.

Chapter 8.

Error Handling and Debugging

1. Try, Catch, Finally Syntax
2. Throwing Custom Errors
3. Debugging Techniques

8 Error Handling and Debugging

8.1 Try, Catch, Finally Syntax

JavaScript's `try...catch...finally` construct provides a structured way to **handle runtime errors gracefully**. It helps your program recover from unexpected issues without crashing.

The Roles of Each Block

- **`try` block:** Contains code that might throw an error during execution.
- **`catch` block:** Handles the error if one occurs in the `try` block. It receives the error object as a parameter.
- **`finally` block:** Contains code that always runs, regardless of whether an error occurred or not—perfect for cleanup.

Basic Example

```
try {
  // Code that may throw an error
  let result = someUndefinedFunction();
  console.log(result);
} catch (error) {
  console.log("An error occurred:", error.message);
} finally {
  console.log("Cleanup or final steps here.");
}
```

Output:

```
An error occurred: someUndefinedFunction is not defined
Cleanup or final steps here.
```

Even though the function was undefined and threw an error, the `catch` block handled it, and the `finally` block executed afterward.

Throwing Errors Inside `try`

You can throw custom errors with the `throw` keyword inside the `try` block:

```
try {
  let age = 15;
  if (age < 18) {
    throw new Error("You must be at least 18 years old.");
  }
  console.log("Access granted.");
} catch (err) {
  console.log("Error:", err.message);
}
```

Output:

```
Error: You must be at least 18 years old.
```

Nested `try...catch`

Sometimes you might nest `try...catch` blocks to handle different error scopes:

```
try {
  try {
    throw new Error("Inner error");
  } catch (innerErr) {
    console.log("Caught inner error:", innerErr.message);
    throw innerErr; // Re-throw to outer catch
  }
} catch (outerErr) {
  console.log("Caught outer error:", outerErr.message);
}
```

Output:

```
Caught inner error: Inner error
Caught outer error: Inner error
```

When Does `finally` Run?

The `finally` block **always runs**, whether an error was caught or not, and even if the `try` or `catch` blocks return or throw errors themselves.

```
function testFinally() {
  try {
    return "Try block";
  } finally {
    console.log("Finally block runs anyway.");
  }
}

console.log(testFinally());
```

Output:

```
Finally block runs anyway.
Try block
```

8.1.1 Summary

- Use **`try`** to wrap code that might fail.
- Use **`catch`** to handle errors and prevent crashes.

- Use **`finally`** for cleanup code that must run regardless.
- You can **throw** errors manually inside `try`.
- **Nested** `try...catch` and **re-throwing** errors help fine-tune error handling.

This structure makes your JavaScript programs robust and easier to debug.

8.2 Throwing Custom Errors

In JavaScript, you can create and throw **custom errors** to provide more meaningful information when something goes wrong. This helps in debugging and controlling program flow effectively.

Creating and Throwing Custom Errors

You throw an error using the `throw` statement, usually with an instance of the built-in `Error` class:

```
function checkAge(age) {
  if (age < 18) {
    throw new Error("Age must be at least 18.");
  }
  console.log("Access granted.");
}

try {
  checkAge(15);
} catch (err) {
  console.log("Error caught:", err.message);
}
```

Output:

```
Error caught: Age must be at least 18.
```

Why Throw Custom Errors?

Throwing errors deliberately helps:

- Enforce business logic or input validation.
- Signal unexpected situations clearly.
- Separate error handling concerns from normal logic flow.

Extending the `Error` Class

For more specific error types, you can create custom error classes by extending the `Error` class:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

function validateEmail(email) {
  if (!email.includes("@")) {
    throw new ValidationError("Invalid email format.");
  }
  console.log("Email is valid.");
}

try {
  validateEmail("invalidEmail");
} catch (err) {
  console.log(err.name + ": " + err.message);
}
```

Output:

```
ValidationError: Invalid email format.
```

Practical Use

Custom errors improve clarity in complex applications by:

- Distinguishing error types via `err.name`.
- Allowing targeted `catch` handling.
- Enhancing debugging with stack traces and custom messages.

8.2.1 Summary

- Use `throw new Error(message)` to raise errors with descriptive messages.
- Create specialized errors by extending `Error`.
- Throwing custom errors enforces rules and improves maintainability.
- Always catch errors to handle them gracefully and keep your app stable.

8.3 Debugging Techniques

Debugging is an essential skill in JavaScript development, helping you find and fix errors or unexpected behavior efficiently. Here are common methods and tools used for debugging:

Using `console.log()`

The simplest way to debug is to insert `console.log()` statements to print variable values or program states:

```
const total = 5 + 10;
console.log("Total:", total);
```

While quick and effective, overusing `console.log()` can clutter your code and console output.

Browser Developer Tools

Modern browsers (Chrome, Firefox, Edge) come with built-in **developer tools** offering powerful debugging features:

- **Console tab:** View logs, errors, and interact with your code.
- **Sources tab:** View source files, set breakpoints, and step through code.

Breakpoints and Stepping Through Code

Set breakpoints to pause execution at a specific line:

- Click the line number in the **Sources** panel.
- When code pauses, use:
  - **Step Over (F10):** Move to the next line.
  - **Step Into (F11):** Enter a called function.
  - **Step Out (Shift+F11):** Exit the current function.

This lets you observe program flow and variable changes line-by-line.

Inspecting Variables and Call Stack

While paused, hover over variables to see their current values or check the **Scope** pane. The **Call Stack** pane shows the sequence of function calls leading to the breakpoint, helping trace how you got there.

Using the `debugger` Statement

You can insert the `debugger` keyword directly in your code:

```
function calculateSum(a, b) {
  debugger; // Execution will pause here if devtools are open
  return a + b;
}
```

When the developer tools are open, execution stops at this line, acting like a breakpoint.

Advanced Techniques

- **Conditional Breakpoints:** Right-click a breakpoint to add a condition (e.g., pause only if `count > 10`), reducing noise during debugging.

- **Exception Breakpoints:** Pause when any error is thrown, catching bugs early.

8.3.1 Practical Debugging Workflow Example

Suppose your function returns incorrect results:

1. Add `console.log()` to check input and intermediate values.
2. If issue persists, set breakpoints in DevTools near suspicious code.
3. Step through the code to watch variable changes.
4. Inspect call stack to understand the execution path.
5. Use conditional breakpoints to focus on edge cases.

8.3.2 Summary

Mastering debugging tools—from simple logs to advanced breakpoints—helps you understand your code's behavior deeply, save time, and write more reliable JavaScript applications.

Chapter 9.

Working with Strings and Regular Expressions

1. String Methods and Properties
2. Template Literals
3. Tagged Template Literals
4. Regular Expressions Syntax and Usage

9 Working with Strings and Regular Expressions

9.1 String Methods and Properties

JavaScript strings come with many built-in methods that allow you to inspect, manipulate, and transform text efficiently. It's important to remember that **strings are immutable**, meaning any method that alters a string actually returns a **new string** without changing the original.

`length` Get String Length

Returns the number of characters in the string.

```
const str = "Hello";
console.log(str.length); // 5
```

`charAt(index)` Get Character at Position

Returns the character at the specified zero-based index.

```
const str = "Hello";
console.log(str.charAt(1)); // "e"
```

`indexOf(substring)` Find First Occurrence

Returns the index of the first occurrence of the substring or `-1` if not found.

```
const str = "Hello";
console.log(str.indexOf("l")); // 2
console.log(str.indexOf("z")); // -1
```

`substring(start, end)` Extract Part of a String

Extracts characters from `start` index up to, but not including, `end` index.

```
const str = "Hello";
console.log(str.substring(1, 4)); // "ell"
```

`slice(start, end)` Similar to `substring`

Works like `substring` but supports negative indexes (counting from the end).

```
const str = "Hello";
console.log(str.slice(1, 4));   // "ell"
console.log(str.slice(-3, -1)); // "ll"
```

`toUpperCase()` and `toLowerCase()` Case Conversion

Return new strings converted to upper or lower case.

```
const str = "Hello";
console.log(str.toUpperCase()); // "HELLO"
console.log(str.toLowerCase()); // "hello"
```

`trim()` Remove Whitespace

Returns a new string with leading and trailing whitespace removed.

```
const padded = "  hello  ";
console.log(padded.trim()); // "hello"
```

`replace(searchValue, newValue)` Replace Substring

Returns a new string with the first match of `searchValue` replaced by `newValue`.

```
const greeting = "Hello world";
console.log(greeting.replace("world", "JavaScript")); // "Hello JavaScript"
```

To replace all occurrences, use a regular expression with the global flag:

```
const text = "foo bar foo";
console.log(text.replace(/foo/g, "baz")); // "baz bar baz"
```

`split(separator)` Split String into Array

Splits a string into an array of substrings using the specified separator.

```
const csv = "red,green,blue";
const colors = csv.split(",");
console.log(colors); // ["red", "green", "blue"]
```

9.1.1 Summary

All string methods return **new strings** and do **not** modify the original string. This immutability means you should always assign or use the returned value. Mastering these methods is essential for effective text processing in JavaScript.

9.2 Template Literals

Template literals are a modern and powerful way to work with strings in JavaScript. They use **backticks** (‘) instead of quotes and allow you to embed expressions and create multiline strings easily.

Syntax and String Interpolation

Inside template literals, you can embed any JavaScript expression using `${}`. This is called **string interpolation** and replaces the traditional cumbersome string concatenation:

```
const name = "Alice";
const age = 25;

const greeting = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(greeting);
// Output: Hello, my name is Alice and I am 25 years old.
```

Multiline Strings

With template literals, you can write strings spanning multiple lines without using escape characters like new line:

```
const multiline = `This is a string
that spans multiple
lines easily.`;
console.log(multiline);
```

Output:

```
This is a string
that spans multiple
lines easily.
```

Embedding Expressions

You can embed any expression inside `${}` — including calculations, function calls, or ternary operators:

```
const a = 5;
const b = 10;

console.log(`Sum of ${a} and ${b} is ${a + b}.`);
// Output: Sum of 5 and 10 is 15.

const age = 25;
console.log(`You are ${age >= 18 ? "an adult" : "a minor"}.`);
// Output: You are an adult.
```

Advantages Over Traditional Concatenation

- Cleaner and more readable code, especially with multiple variables.
- Easier multiline strings without `+` and new line character.
- Direct embedding of complex expressions.

9.2.1 Summary

Template literals simplify string creation and manipulation. Use backticks and `${}` to interpolate variables and expressions, and enjoy natural multiline strings—making your JavaScript code clearer and more expressive.

9.3 Tagged Template Literals

Tagged template literals extend the power of regular template literals by allowing you to **process the template string with a custom function**, called a *tag function*. This function receives the parts of the template string and the interpolated values separately, letting you manipulate or transform them before producing the final output.

How Tag Functions Work

When you prefix a template literal with a function name (the tag), that function is called with:

- An array of literal string parts (the raw text segments between expressions).
- Each interpolated expression as separate arguments.

The tag function can then combine or transform these inputs however it wants.

Basic Example

```
function tag(strings, ...values) {
  console.log("Strings:", strings);
  console.log("Values:", values);
  return strings[0] + values.map((v, i) => v + strings[i + 1]).join("");
}

const name = "Alice";
const age = 25;

const result = tag`Name: ${name}, Age: ${age}`;
console.log(result);
// Output:
// Strings: ["Name: ", ", Age: ", ""]
// Values: ["Alice", 25]
// Name: Alice, Age: 25
```

Use Cases for Tagged Templates

- **Custom formatting:** Add special markup or styling.
- **Sanitization:** Escape user input to prevent injection attacks.
- **Localization:** Format strings differently depending on language or context.

Sanitization Example

```
function sanitize(strings, ...values) {
  return strings.reduce((acc, str, i) => {
    const safeValue = values[i - 1]
      ? String(values[i - 1]).replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;")
      : "";
    return acc + safeValue + str;
  });
}

const userInput = "<script>alert('xss')</script>";
const safeStr = sanitize`User input: ${userInput}`;
console.log(safeStr);
// Output: User input: &lt;script&gt;alert('xss')&lt;/script&gt;
```

9.3.1 Summary

Tagged template literals give you fine-grained control over string creation by passing raw strings and values to a function. This makes them ideal for custom formatting, security sanitization, and dynamic localization. Understanding how to write and use tag functions adds powerful flexibility to your string handling toolkit.

9.4 Regular Expressions Syntax and Usage

Regular expressions (regex) are powerful patterns used to match and manipulate text. In JavaScript, regexes help validate input, search strings, extract data, and perform complex replacements.

Creating Regular Expressions

You can create regexes in two ways:

1. **Regex literal:** Between slashes, e.g. `/pattern/flags`
2. **RegExp constructor:** `new RegExp("pattern", "flags")`

```
const regexLiteral = /hello/i;      // Case-insensitive
const regexConstructor = new RegExp("hello", "i");
```

Common Flags

- `g` — Global search (find all matches)
- `i` — Case-insensitive
- `m` — Multiline mode (`^` and `$` match start/end of lines)

- `u` — Unicode mode (handles Unicode characters correctly)
- `s` — Dotall mode (dot `.` matches newline characters)

Basic Syntax Elements

- **Character classes:** Define sets of characters to match.
  - `[abc]` — Matches any of `a`, `b`, or `c`
  - `\d` — Digit (`[0-9]`)
  - `\w` — Word character (`[a-zA-Z0-9_]`)
  - `\s` — Whitespace (, `\t`, new line characters, etc.)
  - `.` — Any character except newline (or including newline with `s` flag)
- **Quantifiers:** Specify how many times to match:
  - `*` — 0 or more times
  - `+` — 1 or more times
  - `?` — 0 or 1 time
  - `{n}` — Exactly n times
  - `{n,}` — At least n times
  - `{n,m}` — Between n and m times
- **Groups and capturing:**
  - `(abc)` — Capturing group that matches `abc`
  - `(?:abc)` — Non-capturing group
- **Anchors:** Match positions in string.
  - `^` — Start of string (or line with `m` flag)
  - `$` — End of string (or line with `m` flag)
  - `\b` — Word boundary
- **Escape special characters:** Use `\` before `.`, `*`, `?`, `+`, `(`, `)`, `[`, `]`, `{`, `}`, `|`, `^`, `$`, and `\`.

Practical Examples

Testing if a pattern exists

```
const pattern = /hello/i;
console.log(pattern.test("Hello World")); // true
```

Matching all words in a string

```
const text = "Hello there, how are you?";
const words = text.match(/\b\w+\b/g);
console.log(words); // ["Hello", "there", "how", "are", "you"]
```

Searching and replacing

Replace all spaces with dashes:

```
const str = "apple orange banana";
const newStr = str.replace(/\s/g, "-");
console.log(newStr); // "apple-orange-banana"
```

Capturing groups

Extract date parts from `YYYY-MM-DD`:

```
const date = "2023-06-23";
const datePattern = /(\d{4})-(\d{2})-(\d{2})/;
const [, year, month, day] = date.match(datePattern);

console.log(`Year: ${year}, Month: ${month}, Day: ${day}`);
// Year: 2023, Month: 06, Day: 23
```

9.4.1 Summary

Regular expressions are concise, flexible tools for string processing. Understanding their syntax — including flags, character classes, quantifiers, groups, and anchors — unlocks powerful pattern matching capabilities. Practice with common methods like `.test()`, `.match()`, `.replace()`, and `.search()` to harness regex in your JavaScript projects.

Chapter 10.

ES6 and Beyond: Modern JavaScript Syntax

1. Let and Const Revisited
2. Destructuring Assignment (Arrays and Objects)
3. Template Strings and Tagged Templates
4. Enhanced Object Literals (shorthand properties, computed property names)
5. Default, Rest, and Spread Operators
6. Optional Chaining (`?.`) and Nullish Coalescing (`??`)
7. BigInt Syntax and Usage

10 ES6 and Beyond: Modern JavaScript Syntax

10.1 Let and Const Revisited

In modern JavaScript, `let` and `const` provide better ways to declare variables compared to the older `var`. Understanding their differences is key to writing clean, predictable code.

Scope: Block vs. Function

- **`var`** is *function-scoped*, meaning a variable declared with `var` is accessible throughout the entire function it's declared in, regardless of block boundaries like `if` or `for`.
- **`let`** and **`const`** are *block-scoped*, limited to the `{}` block where they are declared.

```
if (true) {
  var x = 1;
  let y = 2;
}
console.log(x); // 1 (var leaks outside block)
console.log(y); // ReferenceError: y is not defined
```

Hoisting and Temporal Dead Zone (TDZ)

All declarations are *hoisted* (moved to the top), but:

- **`var`** is initialized with `undefined` immediately during hoisting, so you can access it before declaration (though it's undefined).
- **`let`** and **`const`** are hoisted but not initialized. Accessing them before declaration causes a *Temporal Dead Zone* error.

```
console.log(aVar);  // undefined
console.log(aLet);  // ReferenceError: Cannot access 'aLet' before initialization

var aVar = 10;
let aLet = 20;
```

Mutability and Reassignment

- **`let`** allows reassignment:

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

- **`const`** prohibits reassignment:

```
const name = "Alice";
name = "Bob"; // TypeError: Assignment to constant variable.
```

However, **`const` does NOT make objects immutable**—you can still modify properties:

```
const obj = { age: 30 };
obj.age = 31; // Allowed
```

Best Practices

- Use **`const` by default** to declare variables that should not be reassigned. This signals intent and reduces bugs.
- Use **`let`** only when you need to reassign values (e.g., loop counters).
- Avoid **`var`** due to its confusing hoisting and scoping behavior.

10.1.1 Summary Example

```
function example() {
  if (true) {
    var x = 5;     // Function-scoped
    let y = 10;    // Block-scoped
    const z = 15;  // Block-scoped, no reassignment
  }
  console.log(x); // 5
  console.log(y); // ReferenceError
  console.log(z); // ReferenceError
}
```

Using `let` and `const` improves readability, scope control, and reduces bugs related to variable hoisting and reassignment in modern JavaScript development.

10.2 Destructuring Assignment (Arrays and Objects)

Destructuring assignment is a concise way to extract values from arrays or properties from objects into distinct variables. It makes your code cleaner and easier to read by reducing repetitive access.

Array Destructuring

Basic syntax extracts values by position:

```
const numbers = [10, 20, 30];
const [a, b, c] = numbers;
console.log(a, b, c); // 10 20 30
```

You can skip elements using commas:

```
const [first, , third] = numbers;
console.log(first, third); // 10 30
```

Default Values

If the extracted value is `undefined`, you can assign a default:

```
const [x = 1, y = 2] = [undefined];
console.log(x, y); // 1 2
```

Nested Destructuring

Arrays can be nested, and destructuring can follow the structure:

```
const nested = [1, [2, 3]];
const [i, [j, k]] = nested;
console.log(i, j, k); // 1 2 3
```

Rest Syntax in Arrays

Collect remaining elements into an array:

```
const [head, ...tail] = [5, 6, 7, 8];
console.log(head); // 5
console.log(tail); // [6, 7, 8]
```

Object Destructuring

Extract properties by matching names:

```
const person = { name: "Alice", age: 30 };
const { name, age } = person;
console.log(name, age); // Alice 30
```

You can assign to variables with different names:

```
const person = { name: "Alice", age: 30 };
const { name: firstName, age: years } = person;
console.log(firstName, years); // Alice 30
```

Default Values and Nested Objects

```
const user = { id: 1, profile: { email: "user@example.com" } };
const { profile: { email, phone = "N/A" } } = user;
console.log(email, phone); // user@example.com N/A
```

Rest Syntax in Objects

Gather remaining properties into a new object:

```
const person = { name: "Alice", age: 30 };
const { name, ...rest } = person;
console.log(name); // Alice
console.log(JSON.stringify(rest,null,2)); // { age: 30 }
```

Common Use Cases

Swapping variables without a temp variable:

```
let a = 1, b = 2;
[a, b] = [b, a];
console.log(a, b); // 2 1
```

Destructuring function parameters:

```
function greet({ name, age }) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}
greet(person); // Hello, Alice. You are 30 years old.
```

10.2.1 Summary

Destructuring assignment allows elegant extraction from arrays and objects with support for default values, nesting, and rest syntax. It simplifies variable assignment, parameter handling, and improves code readability in modern JavaScript.

10.3 Template Strings and Tagged Templates

Template literals are a powerful ES6 feature that allow you to create multi-line strings and embed expressions easily. They use backticks (‘) instead of quotes, improving readability and flexibility compared to traditional string concatenation.

Template Strings Basics

You can embed variables and expressions inside `${}`:

```
const name = "Alice";
const age = 30;

const greeting = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(greeting);
// Output: Hello, my name is Alice and I am 30 years old.
```

They also support multi-line strings naturally, without needing escape characters:

```
const poem = `Roses are red,
Violets are blue,
JavaScript rocks,
And so do you.`;
console.log(poem);
```

Tagged Template Literals

Tagged templates extend template literals by letting you process the string and interpolated values through a custom *tag function*. This function receives the literal parts of the string and the expressions separately, allowing advanced manipulation like formatting, localization, or sanitization.

Syntax:

```
tagFunction`string text ${expression} more text`;
```

How Tag Functions Work

A tag function takes two main arguments:

- An array of string literals (the parts between expressions).
- The evaluated expressions as additional arguments.

Example:

```
function simpleTag(strings, ...values) {
  console.log(strings); // ["Hello, ", " is ", " years old."]
  console.log(values);  // ["Alice", 30]
  return strings[0] + values.map((v, i) => v + strings[i + 1]).join("");
}

const result = simpleTag`Hello, ${name} is ${age} years old.`;
console.log(result);
// Output: Hello, Alice is 30 years old.
```

Practical Tagged Template Use: Sanitization

Prevent unsafe input by escaping special characters:

```
function sanitize(strings, ...values) {
  return strings.reduce((acc, str, i) => {
    const safe = values[i - 1]
      ? String(values[i - 1]).replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;")
      : "";
    return acc + safe + str;
  });
}

const userInput = "<script>alert('xss')</script>";
const safeString = sanitize`User input: ${userInput}`;
console.log(safeString);
// Output: User input: &lt;script&gt;alert('xss')&lt;/script&gt;
```

10.3.1 Summary

Template strings simplify embedding expressions and multi-line strings. Tagged templates add a layer of customization by letting functions process templates before producing the final string, enabling use cases like input sanitization, formatting, or localization—making your string handling both powerful and safe.

10.4 Enhanced Object Literals (shorthand properties, computed property names)

ES6 introduced enhancements to object literal syntax that simplify object creation and increase flexibility. Two key features are **shorthand property names** and **computed property names**.

Shorthand Property Names

When creating an object, if a property name matches the variable name holding its value, you can use the shorthand syntax instead of repeating the name and value.

Traditional syntax:

```
const name = "Alice";
const age = 30;

const person = {
  name: name,
  age: age,
};
console.log(JSON.stringify(person,null,2)); // { name: "Alice", age: 30 }
```

With shorthand:

```
const name = "Alice";
const age = 30;

const person = { name, age };
console.log(JSON.stringify(person,null,2)); // { name: "Alice", age: 30 }
```

This reduces redundancy and makes the code cleaner.

Shorthand Method Definitions

Instead of assigning functions as properties, you can define methods directly inside object literals without the `function` keyword.

Traditional:

```
const obj = {
  greet: function() {
    console.log("Hello!");
  }
};
obj.greet(); // Hello!
```

Shorthand:

```
const obj = {
  greet() {
    console.log("Hello!");
  }
};
obj.greet(); // Hello!
```

Computed Property Names

Computed property names let you use expressions inside square brackets `[]` to dynamically set property keys in object literals.

```
const propName = "score";
const user = {
  [propName]: 100,
};
console.log(user.score); // 100
```

You can use any expression inside the brackets:

```
const prefix = "user";
const id = 42;

const obj = {
  [`${prefix}_${id}`]: "value",
};
console.log(obj.user_42); // value
```

10.4.1 Summary

Enhanced object literals make object creation more concise and dynamic. Shorthand properties and methods reduce boilerplate, improving readability, while computed property names enable dynamic keys, boosting flexibility.

```
const x = 10;
const y = 20;

const point = {
  x,          // shorthand property
  y,
```

```
  move(dx, dy) {   // shorthand method
    this.x += dx;
    this.y += dy;
  },
  [`distance_${x}_${y}`]: Math.sqrt(x*x + y*y) // computed property
};

console.log(point.distance_10_20); // 22.360679774997898
point.move(5, 5);
console.log(JSON.stringify(point,null,2)); // { x: 15, y: 25, move: [Function: move], distance_10_20: 2
```

This syntax keeps your code clean and expressive in modern JavaScript development.

10.5 Default, Rest, and Spread Operators

Modern JavaScript introduces several powerful syntactic features that simplify working with functions, arrays, and objects: **default parameters**, **rest parameters**, and **spread syntax**.

Default Parameters

Default parameters allow you to assign default values to function parameters if no argument or `undefined` is passed:

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

greet("Alice"); // Hello, Alice!
greet();        // Hello, Guest!
```

You can even use previous parameters as defaults:

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

console.log(multiply(5));    // 25 (b defaults to 5)
console.log(multiply(5, 2)); // 10
```

Rest Parameters

Rest parameters collect multiple arguments into a single array, useful for functions with variable numbers of arguments:

```
function sum(...numbers) {
  return numbers.reduce((total, n) => total + n, 0);
}

console.log(sum(1, 2, 3));    // 6
```

```
console.log(sum(4, 5, 6, 7)); // 22
```

Rest parameters must be the last parameter in the function.

Spread Syntax

The spread syntax (...) expands iterable values (like arrays or strings) or object properties into individual elements or key-value pairs.

In function calls:

```
const nums = [1, 2, 3];
console.log(Math.max(...nums)); // 3
```

In arrays:

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

In objects:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const merged = { ...obj1, ...obj2 };
console.log(JSON.stringify(merged,null,2)); // { a: 1, b: 3, c: 4 }
```

Note: Properties in later objects overwrite earlier ones.

10.5.1 Practical Example Combining These Features

```
function createUser(name = "Anonymous", ...roles) {
  return {
    name,
    roles,
    info() {
      console.log(`${this.name} has roles: ${this.roles.join(", ")}`);
    }
  };
}

const user1 = createUser("Alice", "admin", "editor");
user1.info(); // Alice has roles: admin, editor

const user2 = createUser();
user2.info(); // Anonymous has roles:
```

10.5.2 Summary

- **Default parameters** provide fallback values for missing arguments.
- **Rest parameters** gather remaining arguments into arrays.
- **Spread syntax** expands arrays or objects in calls and literals.

Together, they offer flexible, concise ways to manage function arguments and data structures in modern JavaScript.

10.6 Optional Chaining (`?.`) and Nullish Coalescing (`??`)

Modern JavaScript introduces two handy operators — **optional chaining (`?.`)** and **nullish coalescing (`??`)** — that simplify handling `null` or `undefined` values safely and concisely.

Optional Chaining (`?.`)

Optional chaining lets you safely access deeply nested properties or call methods on objects that might be `null` or `undefined` without causing runtime errors.

Instead of:

```
if (user && user.address && user.address.street) {
  console.log(user.address.street);
}
```

You can write:

```
console.log(user?.address?.street);
```

If any part of the chain (`user`, `user.address`) is `null` or `undefined`, the expression short-circuits and returns `undefined` instead of throwing an error.

You can also safely call methods:

```
user?.getName?.();
```

If `getName` exists, it will be called; otherwise, it returns `undefined`.

Nullish Coalescing (`??`)

Nullish coalescing provides a way to assign default values **only when a value is `null` or `undefined`**, unlike the logical OR (`||`) which treats other falsy values like `0`, `""`, or `false` as default triggers.

Example with `||`:

```
const count = 0;
const value = count || 10;
console.log(value); // 10 - but 0 is a valid value!
```

With ??:

```
const count = 0;
const value = count ?? 10;
console.log(value); // 0 - correctly keeps zero
```

Combined Usage Example

```
const config = {
  theme: null,
  user: {
    name: "Alice",
    getAddress() {
      return null;
    }
  }
};

const theme = config.theme ?? "default";       // "default" since theme is null
const street = config.user?.getAddress()?.street ?? "No address"; // "No address"

console.log(theme);  // default
console.log(street); // No address
```

10.6.1 Summary

- Use **optional chaining (`?.`)** to safely access nested properties or call methods without risk of errors when intermediate values are missing.
- Use **nullish coalescing (`??`)** to provide defaults only when dealing with `null` or `undefined`, preserving other falsy but valid values.

Together, they make your code safer, cleaner, and easier to maintain.

10.7 BigInt Syntax and Usage

JavaScript's `BigInt` is a primitive type designed to represent integers larger than the safe integer limit of the standard `Number` type (`Number.MAX_SAFE_INTEGER`, which is 2^53 - 1). It allows you to work safely with arbitrarily large integers without losing precision.

Creating BigInts

You can create a `BigInt` in two ways:

- **Literal syntax:** Append `n` to the end of an integer literal.

```
const bigNum = 9007199254740991n;  // Number.MAX_SAFE_INTEGER as BigInt
const hugeNum = 12345678901234567890123456789n;
```

- **Constructor:** Use the `BigInt()` function with a string or number.

```
const bigFromString = BigInt("9007199254740991999999");
```

Arithmetic Operations

You can perform most standard arithmetic operations with `BigInt` values:

```
const a = 10000000000000000000n;
const b = 90000000000000000000n;

console.log(a + b);  // 190000000000000000000n
console.log(a * 2n); // 200000000000000000000n
console.log(a / 3n); // 3333333333333333333n (integer division)
```

Comparisons and Limitations

`BigInt` values can be compared using standard comparison operators:

```
console.log(10n > 5n);  // true
console.log(10n === 10); // false (different types)
```

Important: You cannot mix `BigInt` and `Number` types directly in arithmetic without explicit conversion:

```
const num = 10;
const big = 10n;

console.log(big + BigInt(num)); // 20n (works)
console.log(big + num);          // TypeError: Cannot mix BigInt and other types
```

Use `BigInt()` or `Number()` conversions to handle this.

10.7.1 Summary

`BigInt` extends JavaScript's numeric capabilities to safely handle very large integers beyond the limitations of `Number`. Use the `n` literal or `BigInt()` constructor to create values, and keep in mind that arithmetic must be between values of the same type. This powerful feature is essential for applications dealing with cryptography, large IDs, or high-precision calculations.

```
const bigValue = 9007199254740993n;
console.log(bigValue + 1n);  // 9007199254740994n
```

Chapter 11.

Classes and Prototypes

1. Introduction to Prototypes
2. Constructor Functions
3. ES6 Classes Syntax
4. Private Class Fields and Methods (`#privateField`)
5. Inheritance and Subclasses
6. Static Methods and Properties

11 Classes and Prototypes

11.1 Introduction to Prototypes

JavaScript uses a **prototype-based inheritance** model, which means that objects inherit properties and methods from other objects, called *prototypes*, rather than from classes (as in classical OOP languages). Understanding prototypes is fundamental to mastering JavaScript's object system.

Prototype Chains and Property Lookup

Every JavaScript object has an internal link to another object called its **prototype** (accessible via `[[Prototype]]` or in code via `Object.getPrototypeOf(obj)` or the `__proto__` property). When you try to access a property on an object:

1. JavaScript first looks for the property on the *own properties* of the object itself.
2. If not found, it follows the prototype link and searches the prototype object.
3. This lookup continues up the **prototype chain** until the property is found or the chain ends at `null`.

This delegation mechanism allows objects to share methods and properties efficiently.

Own Properties vs. Inherited Properties

- **Own properties** are properties that exist directly on the object.
- **Inherited properties** come from the object's prototype or prototypes higher in the chain.

You can check if a property is an own property using:

```
const obj = { a: 1 };
console.log(obj.hasOwnProperty('a')); // true
console.log(obj.hasOwnProperty('toString')); // false, inherited from prototype
```

Prototype Object

When you create an object using an object literal or `new Object()`, it inherits from `Object.prototype`. This prototype provides methods like `toString()`, `hasOwnProperty()`, etc.

```
const obj = { name: "Alice" };
console.log(obj.toString()); // [object Object]
```

Here, `toString` is not on `obj` itself, but inherited from `Object.prototype`.

Visualizing the Prototype Chain

```
obj --> Object.prototype --> null
```

If you create an object via a constructor function, the prototype chain looks like:

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

Person.prototype.greet = function() {
  console.log(`Hello, ${this.name}`);
};

const alice = new Person("Alice");
alice.greet();  // "Hello, Alice"
```

Here, `alice` has own property `name`, and inherits `greet` from `Person.prototype`. The full chain:

```
alice --> Person.prototype --> Object.prototype --> null
```

Practical Example: Accessing Properties

```
console.log(alice.name);       // "Alice" (own property)
console.log(alice.greet);      // function (inherited)
console.log(alice.toString()); // function (inherited from Object.prototype)
```

If you delete an own property, access falls back to prototype properties if available:

```
delete alice.name;
console.log(alice.name); // undefined (no property on prototype)
```

11.1.1 Summary

- Every object has a prototype from which it inherits properties.
- Property lookup follows the prototype chain until a match is found or chain ends.
- Own properties belong directly to the object; inherited properties come from prototypes.
- Prototypes enable efficient code reuse and are the basis of JavaScript's inheritance system.

Understanding prototypes lays the groundwork for mastering constructor functions, ES6 classes, and inheritance patterns.

11.2 Constructor Functions

Before ES6 introduced `class` syntax, JavaScript used **constructor functions** as the primary pattern for creating multiple similar objects with shared behavior. Constructor functions work together with prototypes to enable inheritance and method sharing.

Defining a Constructor Function

A constructor function is simply a regular function, but by convention, its name starts with a capital letter to distinguish it from normal functions. When called with the `new` keyword, it creates a new object, sets up the prototype linkage, and binds `this` to the new object.

```
function Person(name, age) {
  this.name = name;  // own property
  this.age = age;
}
```

Using `new`

Calling a constructor function with `new` does the following:

1. Creates a new empty object.
2. Sets the new object's internal prototype (`[[Prototype]]`) to the constructor's `.prototype` property.
3. Binds `this` inside the constructor to the new object.
4. Executes the constructor code.
5. Returns the new object (unless the constructor returns a different object explicitly).

```
const alice = new Person("Alice", 30);
console.log(alice.name); // "Alice"
```

Adding Methods to Prototypes

To avoid duplicating methods for every instance (which wastes memory), methods are usually added to the constructor's `prototype` object. All instances then share these methods via the prototype chain.

```
Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}.`);
};

alice.greet();  // "Hello, my name is Alice."
```

This way, all objects created by `Person` share the same `greet` method.

Prototype Chain Visualization

```
alice --> Person.prototype --> Object.prototype --> null
```

`alice` inherits `greet` from `Person.prototype` and default methods like `toString` from

`Object.prototype`.

Example Putting It All Together

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

Person.prototype.greet = function() {
  console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
};

const bob = new Person("Bob", 25);
bob.greet();  // Hi, I'm Bob and I'm 25 years old.
```

11.2.1 Summary

Constructor functions combined with prototypes form the classical way to create reusable objects in JavaScript. The `new` keyword handles object creation and prototype assignment, while methods on the constructor's prototype enable efficient method sharing across instances. Understanding this pattern provides a solid foundation for the ES6 class syntax introduced later.

11.3 ES6 Classes Syntax

In ES6, JavaScript introduced the `class` syntax to provide a clearer, more concise way to create constructor functions and work with prototypes. While **classes are syntactic sugar** over the existing prototype-based system, they make object-oriented programming more intuitive and easier to write.

Class Declaration and Constructor

A class declaration uses the `class` keyword followed by the class name. Inside, a special method named `constructor` initializes new objects when the class is instantiated with `new`.

```
class Person {
  constructor(name, age) {
    this.name = name;  // own property
    this.age = age;
  }
}
```

Using `new` with this class creates an instance:

```
const alice = new Person('Alice', 30);
console.log(alice.name); // "Alice"
```

Defining Methods

Methods inside a class declaration are automatically added to the class's prototype, so all instances share them:

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

  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}

const bob = new Person('Bob', 25);
bob.greet();  // "Hello, my name is Bob."
```

This is equivalent to adding methods to the prototype manually in constructor functions, but the syntax is cleaner and more readable.

Comparison: Class vs Constructor Function

```
// Constructor function pattern
function PersonCF(name, age) {
  this.name = name;
  this.age = age;
}
PersonCF.prototype.greet = function() {
  console.log(`Hi, I'm ${this.name}`);
};

// ES6 class syntax
class PersonClass {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
}

const p1 = new PersonCF('Charlie', 40);
const p2 = new PersonClass('Dana', 35);
p1.greet(); // Hi, I'm Charlie
p2.greet(); // Hi, I'm Dana
```

How `new` Works with Classes

When you call a class with `new`, it behaves similarly to constructor functions:

- Creates a new object.
- Sets the new object's prototype to the class's `prototype` property.
- Binds `this` inside the constructor to the new object.
- Returns the new object.

Trying to call a class without `new` results in a `TypeError`, enforcing the proper usage:

```
const instance = PersonClass('Eve', 28);  // TypeError: Class constructor cannot be invoked without 'ne
```

11.3.1 Advantages of ES6 Classes

- **Cleaner, declarative syntax** that resembles class-based languages.
- **Built-in inheritance support** via `extends` and `super`.
- Methods are automatically added to prototypes.
- Classes are not hoisted like function declarations, encouraging better coding practices.

11.3.2 Summary

ES6 classes simplify prototype-based inheritance with familiar class syntax. They enhance code clarity while retaining JavaScript's dynamic prototype chain under the hood, making object-oriented programming more accessible and maintainable.

11.4 Private Class Fields and Methods (`#privateField`)

ES2022 introduced **private class fields and methods** in JavaScript, allowing true encapsulation inside classes. These private members are prefixed with a `#` and are only accessible **within the class body**, helping to protect internal state from external access or modification.

Syntax and Semantics

To declare a private field or method, prepend its name with `#`:

```
class Counter {
  #count = 0;               // private field

  #increment() {            // private method
    this.#count++;
```

```
  }

  incrementAndShow() {
    this.#increment();
    console.log(this.#count);
  }
}

const c = new Counter();
c.incrementAndShow(); // 1
// c.#count;          // SyntaxError: Private field '#count' must be declared in an enclosing class
// c.#increment();    // SyntaxError: Private field '#increment' must be declared in an enclosing class
```

Private fields are not properties on the instance object itself and cannot be accessed or modified outside the class, even through reflection or bracket notation.

Benefits of Private Fields

- **True encapsulation**: Unlike previous patterns using closures or naming conventions (`_private`), private fields are enforced by the language.
- **Improved security and maintainability**: Internal state is hidden, preventing accidental external changes.
- **Cleaner API design**: Consumers of your class only see the intended public interface.

Limitations

- Private fields are **not accessible** outside the class, even by subclasses.
- You **cannot dynamically access** private members using bracket notation.
- They are not enumerable and do not appear in `for...in` loops or `Object.keys()`.

Practical Example

```
class BankAccount {
  #balance = 0;

  deposit(amount) {
    if (amount > 0) {
      this.#balance += amount;
    }
  }

  withdraw(amount) {
    if (amount > 0 && amount <= this.#balance) {
      this.#balance -= amount;
    } else {
      console.log('Insufficient funds');
    }
  }

  getBalance() {
    return this.#balance;
  }
}
```

```
const account = new BankAccount();
account.deposit(100);
account.withdraw(30);
console.log(account.getBalance()); // 70
// console.log(account.#balance); // SyntaxError
```

11.4.1 Summary

Private class fields and methods using `#` provide built-in encapsulation in JavaScript classes. They help create robust, secure, and maintainable code by restricting access to internal details, a crucial feature for modern object-oriented programming.

11.5 Inheritance and Subclasses

JavaScript supports class-based inheritance through the `extends` keyword, which allows one class to inherit properties and methods from another. This enables **code reuse**, **specialization**, and the creation of class hierarchies.

Using `extends` and `super()`

To create a subclass, use the `extends` keyword:

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

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Call parent constructor
    this.breed = breed;
  }

  speak() {
    console.log(`${this.name} barks. (${this.breed})`);
  }
}
```

The `super()` function must be called **before accessing `this`** in a subclass constructor. It invokes the parent class's constructor.

Inheriting and Overriding

Subclass instances inherit all public methods and properties of the superclass unless overridden. You can also call the superclass's methods explicitly using `super.methodName()`.

```
const dog = new Dog("Rex", "Labrador");
dog.speak(); // Rex barks. (Labrador)
```

If a subclass doesn't define its own constructor, JavaScript will automatically use the parent's constructor with all arguments.

Extending Behavior

You can extend or enhance superclass behavior by calling `super.method()` within an overridden method:

```
class Bird extends Animal {
  speak() {
    super.speak(); // Call parent method
    console.log(`${this.name} chirps.`);
  }
}

const robin = new Bird("Robin");
robin.speak();
// Output:
// Robin makes a sound.
// Robin chirps.
```

Inheritance Hierarchy

Classes can be extended multiple levels deep:

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

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}
class Mammal extends Animal {
  hasFur() {
    return true;
  }
}

class Cat extends Mammal {
  speak() {
    console.log(`${this.name} meows.`);
  }
}

const kitty = new Cat("Whiskers");
```

```
kitty.speak();     // Whiskers meows.
console.log(kitty.hasFur()); // true
```

11.5.1 Summary

Inheritance via `extends` and `super()` allows JavaScript classes to build upon existing ones. Subclasses can override or enhance behavior, making your code more modular and easier to maintain. Always use `super()` when initializing base class state in subclass constructors, and consider method overriding thoughtfully to preserve clarity and correctness.

11.6 Static Methods and Properties

In JavaScript, **static methods and properties** belong to the class **itself**, not to instances of the class. These are useful for utility functions, configuration values, or operations that don't depend on individual object state.

Defining Static Members

Use the `static` keyword to define static methods or properties:

```
class MathUtils {
  static PI = 3.14159;

  static square(x) {
    return x * x;
  }

  static circleArea(radius) {
    return MathUtils.PI * MathUtils.square(radius);
  }
}
```

You can access static members directly on the class:

```
console.log(MathUtils.square(5));        // 25
console.log(MathUtils.circleArea(3));    // 28.27431
console.log(MathUtils.PI);               // 3.14159
```

Trying to access a static method or property on an instance will result in `undefined`:

```
const utils = new MathUtils();
console.log(utils.square); // undefined
```

Use Cases

Static methods are ideal for:

- **Utility functions** (e.g., `Math.random()` or `Array.isArray()`)
- **Class-level operations** like validation or comparisons
- **Constants** that are shared and do not change per instance

Example of a static factory method:

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

  static createGuest() {
    return new User("Guest");
  }
}

const guest = User.createGuest();
console.log(guest.name); // Guest
```

11.6.1 Summary

Static members are powerful for encapsulating functionality at the class level. They keep code organized and separate from instance-specific logic. Use `static` when the behavior or data doesn't require access to `this` or instance properties.

Chapter 12.

Modules and Import/Export

1. Module Syntax Overview
2. Named and Default Exports
3. Importing Modules
4. Dynamic Imports

12 Modules and Import/Export

12.1 Module Syntax Overview

JavaScript modules are a way to split code into separate files, each with its own scope. Introduced in **ES6 (ECMAScript 2015)**, modules promote **code reuse**, **maintainability**, and **encapsulation**. With modules, developers can organize large codebases by separating logic into smaller, manageable parts.

In traditional JavaScript (pre-ES6), all code shared the same global scope unless wrapped in an IIFE or similar pattern. This made it easy to accidentally overwrite variables or functions and created dependency management challenges. ES6 modules solve these issues by providing a built-in syntax for **importing** and **exporting** functionality across files.

Key Benefits of Modules:

- Avoids polluting the global scope
- Encourages separation of concerns
- Makes code more reusable and testable
- Supports static analysis and tree shaking (removing unused code during bundling)

Basic Module Syntax

To export a value from a module file:

```
// mathUtils.js
export const add = (a, b) => a + b;
export const PI = 3.14;
```

To use the exported code in another file:

```
// app.js
import { add, PI } from './mathUtils.js';

console.log(add(2, 3)); // 5
console.log(PI);        // 3.14
```

Each module has its **own scope**, so variables and functions defined inside aren't accessible unless explicitly exported. Modules are **strict mode** by default.

How Modules Are Loaded

In modern environments (like browsers and Node.js), modules must be included with `type="module"` in HTML:

```
<script type="module" src="app.js"></script>
```

Unlike older `<script>` tags, modules are **deferred by default** (they wait for the HTML to be parsed before running) and **loaded asynchronously**, which helps improve performance.

Summary

ES6 modules bring structure and clarity to JavaScript development. By enabling clean, scoped, and reusable code across files, they have become the standard for building modern JavaScript applications.

12.2 Named and Default Exports

JavaScript modules allow you to export variables, functions, or classes from one file and import them in another. ES6 provides two main export types: **named exports** and **default exports**. Understanding both is essential for writing modular, reusable code.

12.2.1 Named Exports

Named exports allow you to export **multiple values** from a file by name. They must be imported using the exact same identifier.

Exporting named values:

```
// mathUtils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
export const PI = 3.14159;
```

Importing named values:

```
// app.js
import { add, PI } from './mathUtils.js';

console.log(add(2, 3)); // 5
console.log(PI);        // 3.14159
```

You can also **rename** imports:

```
import { subtract as minus } from './mathUtils.js';
console.log(minus(5, 2)); // 3
```

12.2.2 Default Exports

A module can have **only one default export**, which doesn't require curly braces when importing.

Exporting a default value:

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

Importing the default export:

```
// main.js
import greet from './greeter.js';
console.log(greet('Alice')); // Hello, Alice!
```

You can also export classes or objects by default:

```
// user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}
```

12.2.3 Combining Named and Default Exports

You can mix named and default exports in the same file:

```
// config.js
export const theme = 'dark';
export default function loadConfig() {
  return { version: '1.0.0' };
}
```

```
// app.js
import loadConfig, { theme } from './config.js';
console.log(theme);        // 'dark'
console.log(loadConfig()); // { version: '1.0.0' }
```

12.2.4 Summary

Use **named exports** when you want to export multiple related items and **default exports** when a module provides a single main feature. Choosing the right export style improves clarity and consistency across your codebase.

12.3 Importing Modules

JavaScript's ES6 module system allows you to **import code from other files** using the `import` keyword. Modules improve organization, reuse, and readability. There are several import styles depending on how the module exports its contents—**named**, **default**, or a combination of both.

12.3.1 Named Imports

Use curly braces `{}` to import specific items by name:

```
// math.js
export const add = (a, b) => a + b;
export const multiply = (a, b) => a * b;
```

```
// app.js
import { add, multiply } from './math.js';
console.log(add(2, 3));      // 5
console.log(multiply(4, 5)); // 20
```

You can also **rename** named imports using `as`:

```
import { multiply as times } from './math.js';
console.log(times(3, 3)); // 9
```

12.3.2 Default Imports

Modules can export one **default** item, which is imported without braces:

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

```
// main.js
import greet from './greet.js';
console.log(greet('Alice')); // Hello, Alice!
```

12.3.3 Mixed Imports

If a module has both default and named exports, you can import both:

```
// config.js
export const version = '1.0.0';
export default function loadConfig() {
  return { debug: true };
}
```

```
import loadConfig, { version } from './config.js';
console.log(loadConfig()); // { debug: true }
console.log(version);      // 1.0.0
```

12.3.4 Importing Everything

You can import *all* exports as a single object using `* as`:

```
import * as MathUtils from './math.js';
console.log(MathUtils.add(1, 2)); // 3
console.log(MathUtils.multiply(2, 5)); // 10
```

12.3.5 Destructuring on Imports?

While import syntax resembles destructuring, it's not the same. You cannot use typical object destructuring like this:

```
// NO Invalid
const { add } = import './math.js';
```

Instead, use standard import syntax.

12.3.6 Static vs. Dynamic Imports

So far we've seen **static imports**, which are evaluated at load time. For **dynamic loading**, use `import()`:

```
import('./math.js').then(module => {
  console.log(module.add(1, 1)); // 2
});
```

This is useful for lazy loading or conditional module loading.

12.3.7 Summary

Mastering import styles—named, default, mixed, and dynamic—lets you organize and use code efficiently across modules. Choose the style that best fits your module's structure and your app's needs.

12.4 Dynamic Imports

In addition to static `import` statements, JavaScript supports **dynamic imports** using the `import()` function. Unlike static imports, which are hoisted and must appear at the top of a module, `import()` is a **function-like expression** that loads modules asynchronously at runtime and returns a **Promise**.

12.4.1 Syntax

```
import(moduleSpecifier)
  .then((module) => {
    // Use the imported module here
  })
  .catch((err) => {
    // Handle loading errors
  });
```

Or with `async/await`:

```
async function loadModule() {
  try {
    const module = await import('./utils.js');
    module.doSomething();
  } catch (err) {
    console.error('Failed to load module:', err);
  }
}
```

12.4.2 Use Cases

Dynamic imports are ideal for:

- **Code splitting**: Load only the code you need when you need it.
- **Lazy loading**: Defer loading expensive logic until user interaction.
- **Conditional imports**: Load modules based on runtime conditions.
- **Progressive enhancement**: Load optional features in supported environments.

12.4.3 Example: Lazy Load a Helper

```
// Only load helper if needed
const btn = document.querySelector('#loadButton');
btn.addEventListener('click', async () => {
  const { helper } = await import('./helper.js');
  helper();
});
```

12.4.4 Example: Conditional Loading

```
if (navigator.language === 'fr') {
  import('./locale-fr.js').then(({ translate }) => {
    translate();
  });
}
```

12.4.5 Notes

- Modules loaded via `import()` are **cached**, just like static imports.
- The module path must be a **string or a template literal**—you can construct it dynamically if needed.
- Works only in **module-supporting environments** (modern browsers, Node.js ESM mode).

Dynamic imports give you flexibility to load code on demand, improving performance and responsiveness in modern JavaScript applications.

Chapter 13.

Asynchronous JavaScript

1. Callbacks and Callback Hell
2. Promises and Promise Chaining
3. Async/Await Syntax
4. Error Handling in Async Code

13 Asynchronous JavaScript

13.1 Callbacks and Callback Hell

In JavaScript, **callbacks** are functions passed as arguments to other functions to be executed later—often after an asynchronous operation completes. This pattern has been foundational to handling asynchronous behavior in JavaScript, such as responding to user actions or handling network responses.

13.1.1 Basic Callback Example

```
function greetUser(name, callback) {
  console.log(`Hello, ${name}`);
  callback();
}

greetUser('Alice', () => {
  console.log('Callback executed after greeting.');
});
```

In this example, the callback runs *after* the greeting message. Callbacks like this are commonly used in asynchronous tasks such as timers or event listeners.

13.1.2 Asynchronous Callback Example

```
setTimeout(() => {
  console.log('This runs after 2 seconds');
}, 2000);
```

Here, the function inside `setTimeout` is a callback that executes after a delay, demonstrating asynchronous behavior.

13.1.3 Callback Hell

When callbacks are deeply nested—such as in sequential asynchronous operations—code becomes difficult to read and maintain. This issue is often called **callback hell** or the **pyramid of doom**.

```
getData((data) => {
  processData(data, (processed) => {
    saveData(processed, (response) => {
      console.log('Data saved:', response);
    });
  });
});
```

As shown above, each function waits for the previous one to complete. This nesting quickly becomes unwieldy in complex workflows, making debugging and error handling challenging.

13.1.4 Real-World Example

```
document.querySelector('#myBtn').addEventListener('click', () => {
  setTimeout(() => {
    console.log('Button clicked, delayed action');
  }, 1000);
});
```

This combines an event-driven callback (button click) with a timer-based callback, a pattern common in web applications.

13.1.5 Limitations of Callbacks

- Hard to manage when chaining multiple operations.
- Error handling becomes messy and inconsistent.
- Encourages deeply nested, hard-to-read code.

To address these problems, JavaScript introduced **Promises** and later **async/await** to make asynchronous code more readable and maintainable—topics we'll explore in the following sections.

13.2 Promises and Promise Chaining

Promises are a modern JavaScript feature designed to simplify handling asynchronous operations. They offer a cleaner, more readable alternative to deeply nested callbacks.

A **Promise** represents a value that may be available now, later, or never. It has three **states**:

- `pending`: the initial state, not yet fulfilled or rejected
- `fulfilled`: the operation completed successfully

- `rejected`: the operation failed

13.2.1 Creating a Promise

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve('Data loaded');
      } else {
        reject('Error loading data');
      }
    }, 1000);
  });
};
```

In this example, `fetchData` simulates an asynchronous task. After 1 second, it either **resolves** or **rejects** the promise.

13.2.2 Handling with `.then()` and `.catch()`

```
fetchData()
  .then((result) => {
    console.log('Success:', result);
  })
  .catch((error) => {
    console.error('Failure:', error);
  })
  .finally(() => {
    console.log('Finished');
  });
```

- `.then()` handles the fulfilled result.
- `.catch()` handles any rejection.
- `.finally()` always runs, regardless of success or failure.

13.2.3 Promise Chaining

Chaining allows you to sequence multiple async operations without nesting.

```
const step1 = () => Promise.resolve('Step 1 complete');
const step2 = (msg) => Promise.resolve(`${msg} → Step 2 complete`);
const step3 = (msg) => Promise.resolve(`${msg} → Step 3 complete`);

step1()
  .then((res1) => step2(res1))
  .then((res2) => step3(res2))
  .then((final) => console.log('All done:', final));
```

This produces:

```
All done: Step 1 complete → Step 2 complete → Step 3 complete
```

Each `.then()` returns a new promise, allowing results to be passed along the chain in a readable flow.

13.2.4 Error Handling in Chains

Errors can be caught anywhere in the chain:

```
Promise.resolve('Start')
  .then(() => {
    throw new Error('Oops');
  })
  .then(() => {
    console.log('This will not run');
  })
  .catch((err) => {
    console.error('Caught error:', err.message);
  });
```

13.2.5 Summary

Promises simplify asynchronous programming by avoiding deeply nested callbacks and enabling readable sequencing through chaining. Combined with `.catch()` and `.finally()`, they provide a powerful, consistent way to handle async workflows. In the next section, we'll look at `async/await`—a further improvement in asynchronous syntax.

13.3 Async/Await Syntax

The `async` and `await` keywords provide a powerful and readable way to work with asynchronous code in JavaScript. They are syntactic sugar built on top of Promises, allowing you

to write async operations in a style that resembles synchronous code—making logic easier to follow and debug.

13.3.1 Defining an Async Function

An `async` function always returns a Promise, even if it appears to return a plain value:

```
async function greet() {
  return 'Hello!';
}

greet().then(console.log); // Output: Hello!
```

You can also use arrow function syntax:

```
const greet = async () => 'Hello!';
```

13.3.2 Awaiting Promises

The `await` keyword pauses the execution of an `async` function until a Promise resolves (or rejects), letting you write code in a top-down, readable style:

```
const fetchData = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve('Data received'), 1000);
  });
};

async function load() {
  console.log('Loading...');
  const data = await fetchData();
  console.log(data);
  console.log('Done');
}

load();
```

Output:

```
Loading...
(Data arrives after 1 second)
Data received
Done
```

This avoids the need for `.then()` chaining and nested callbacks.

13.3.3 Handling Errors with try/catch

When using `await`, you can wrap calls in `try...catch` blocks to handle errors cleanly:

```
const fetchWithError = () => {
  return new Promise((_, reject) => {
    setTimeout(() => reject('Fetch failed'), 1000);
  });
};

async function safeLoad() {
  try {
    const data = await fetchWithError();
    console.log(data);
  } catch (err) {
    console.error('Error:', err);
  }
}

safeLoad();
```

13.3.4 Summary

`async` and `await` streamline asynchronous programming, improving clarity and error management. By eliminating `.then()` and `.catch()` chains, they help developers write logic in a more synchronous and readable way—while still leveraging the full power of Promises. Use them for clean, maintainable async workflows.

13.4 Error Handling in Async Code

Handling errors in asynchronous JavaScript is critical for building robust and resilient applications. Since async operations execute outside the standard call stack, traditional `try...catch` doesn't always work—so you need specific strategies for callbacks, Promises, and `async/await`.

13.4.1 Callbacks

In callback-based code, error handling is usually manual. The convention is to pass the error as the first argument:

```
function asyncTask(callback) {
  setTimeout(() => {
```

```
    const error = true;
    if (error) {
      callback(new Error('Something went wrong'), null);
    } else {
      callback(null, 'Success!');
    }
  }, 500);
}

asyncTask((err, result) => {
  if (err) {
    console.error('Error:', err.message);
  } else {
    console.log(result);
  }
});
```

This pattern can become hard to manage, especially with nested callbacks (i.e., "callback hell").

13.4.2 Promises

With Promises, errors are propagated via `.catch()`:

```
function fetchData() {
  return new Promise((_, reject) => {
    setTimeout(() => reject('Fetch failed'), 500);
  });
}

fetchData()
  .then(data => console.log(data))
  .catch(error => console.error('Caught:', error));
```

A single `.catch()` at the end of a chain can handle any error thrown in the previous `.then()` blocks.

13.4.3 Async/Await with try...catch

When using `async/await`, wrap `await` calls in `try...catch` blocks to handle errors:

```
async function loadData() {
  try {
    const data = await fetchData();
    console.log(data);
  } catch (err) {
    console.error('Async error:', err);
```

```
  }
}
```

This structure makes error handling more readable and aligns closely with synchronous error management.

13.4.4 Summary

- Use error-first conventions with callbacks.
- Handle Promises with `.catch()` or `.finally()`.
- Use `try...catch` blocks in `async` functions for clean, synchronous-style error handling.

Properly catching and responding to errors ensures your async code fails gracefully and predictably.

Chapter 14.

Generators and Iterators

1. Iterators Protocol
2. Generator Functions and `yield`
3. Using Generators for Async Control Flow

14 Generators and Iterators

14.1 Iterators Protocol

In JavaScript, the **iterator protocol** defines a standard way for objects to produce a sequence of values, one at a time, allowing them to be iterated over using loops like `for...of`.

What is an Iterator?

An **iterator** is an object that provides access to a sequence of values. It must have a `next()` method that returns an object with two properties:

- `value`: the current value in the sequence.
- `done`: a boolean indicating whether the sequence is finished (`true`) or if more values remain (`false`).

Each call to `next()` advances the iterator and returns the next item.

Iterable Objects and `Symbol.iterator`

An **iterable** is an object that implements the `Symbol.iterator` method, which returns an iterator. This method allows objects to work with JavaScript constructs like `for...of`, spread syntax, and array destructuring.

How Iteration Works Flow Overview

1. The iterable's `Symbol.iterator` method is called to get an iterator.
2. The iterator's `next()` method is called repeatedly.
3. Each call returns `{ value, done }`.
4. When `done` is `true`, the iteration stops.

Example: Simple Custom Iterator

```
const myIterable = {
  data: [10, 20, 30],
  [Symbol.iterator]() {
    let index = 0;
    const data = this.data;
    return {
      next() {
        if (index < data.length) {
          return { value: data[index++], done: false };
        } else {
          return { value: undefined, done: true };
        }
      }
    };
  }
};

// Using the custom iterator with for...of:
for (const num of myIterable) {
```

```
  console.log(num); // 10, then 20, then 30
}
```

Diagram Explanation:

```
[Iterable Object]
      |
      | Symbol.iterator() called
      v
[Iterator Object]
      |
      | next() called
      v
{ value, done } - value returned until done = true
```

By implementing the iterator protocol, objects can seamlessly integrate with JavaScript's iteration mechanisms, providing powerful ways to handle sequences of data with custom logic.

14.2 Generator Functions and `yield`

Generator functions are a special type of function in JavaScript that can pause execution and resume later, producing a sequence of values over time. They are declared using the `function*` syntax and utilize the `yield` keyword to pause and send values back to the caller.

What is a Generator Function?

A **generator function** looks like a regular function but is declared with an asterisk (`*`) after the `function` keyword:

```
function* myGenerator() {
  // code
}
```

Unlike normal functions that run to completion when called, generators **pause execution each time they encounter a `yield` statement** and resume when their `.next()` method is called again.

How `yield` Works

- The `yield` keyword pauses the generator and sends a value back to the caller.
- Calling `.next()` on the generator resumes execution from where it was paused.
- Each call to `.next()` returns an object: `{ value, done }`.
  - `value` is the yielded value.

– `done` is `true` when the generator has finished.

Example: Simple Number Sequence Generator

```
function* countUpTo(limit) {
  let count = 1;
  while (count <= limit) {
    yield count;  // Pause and output current count
    count++;
  }
}

const counter = countUpTo(3);

console.log(JSON.stringify(counter.next(),null,2)); // { value: 1, done: false }
console.log(JSON.stringify(counter.next(),null,2)); // { value: 2, done: false }
console.log(JSON.stringify(counter.next(),null,2)); // { value: 3, done: false }
console.log(JSON.stringify(counter.next(),null,2)); // { value: undefined, done: true }
```

Using Generators in `for...of` Loop

Generators implement the iterator protocol, so you can loop through their yielded values easily:

```
for (const num of countUpTo(5)) {
  console.log(num); // Logs 1, 2, 3, 4, 5
}
```

Practical Use Case: Infinite Sequence Generator

Generators can produce infinite sequences without blocking:

```
function* infiniteSequence() {
  let i = 0;
  while (true) {
    yield i++;
  }
}

const seq = infiniteSequence();

console.log(seq.next().value); // 0
console.log(seq.next().value); // 1
console.log(seq.next().value); // 2
```

14.2.1 Summary

Generator functions let you write code that produces values on demand, pausing and resuming execution. This behavior makes them ideal for tasks like:

- Lazy evaluation
- Handling asynchronous workflows (covered later)
- Creating iterators without complex state management

By combining `function*` and `yield`, JavaScript offers powerful control over iteration and execution flow.

14.3 Using Generators for Async Control Flow

Before `async/await` became widely adopted, **generators** were used as a powerful tool to manage asynchronous control flow in JavaScript. By yielding Promises from generator functions, libraries like **co** could pause execution until a Promise resolved, allowing asynchronous code to be written in a synchronous style.

How Generators Help with Async Code

Normally, handling asynchronous operations requires callbacks or chaining Promises, which can get complicated. Generators combined with Promises allow you to write **linear, step-by-step code** that pauses at each asynchronous operation until it completes.

A generator function `yield`s a Promise, and the controlling code waits for it to resolve before continuing execution with the resolved value.

Manual Example: Generator Promise Runner

Here's a simple manual implementation to run a generator that yields Promises sequentially:

```
function run(generatorFunc) {
  const iterator = generatorFunc();

  function handle(result) {
    if (result.done) return Promise.resolve(result.value);

    return Promise.resolve(result.value).then(res => handle(iterator.next(res)));
  }

  return handle(iterator.next());
}

// Example async function returning a Promise
function wait(ms) {
  return new Promise(resolve => setTimeout(() => resolve(`Waited ${ms} ms`), ms));
}

// Generator function yielding Promises
function* asyncTasks() {
  const msg1 = yield wait(1000);
  console.log(msg1); // Waited 1000 ms

  const msg2 = yield wait(500);
```

```
  console.log(msg2); // Waited 500 ms

  return "All done!";
}

run(asyncTasks).then(finalMsg => console.log(finalMsg));
```

How This Works

- The `run` function starts the generator.
- Each yielded Promise is awaited (via `.then`).
- Once resolved, the generator resumes with the resolved value.
- This continues until the generator completes.

Libraries Using This Pattern

The **co** library popularized this approach:

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

co(function* () {
  const res1 = yield wait(1000);
  console.log(res1);

  const res2 = yield wait(500);
  console.log(res2);
});
```

This made asynchronous flows easier to write and maintain before `async/await`.

14.3.1 Summary

Using generators to control async flow is an elegant solution that leverages `yield` to pause for Promises. This approach enabled sequential async logic with readable, synchronous-looking code — a major stepping stone in JavaScript's evolution toward `async/await`.

Chapter 15.

Symbols and Well-Known Symbols

1. Creating and Using Symbols
2. Symbol Properties and Use Cases
3. Well-Known Symbols

15 Symbols and Well-Known Symbols

15.1 Creating and Using Symbols

Symbols are a new primitive type introduced in ES6 that represent **unique, immutable identifiers**. Unlike strings or numbers, each Symbol is guaranteed to be unique, even if created with the same description. This uniqueness makes Symbols especially useful as keys for object properties, preventing accidental property name collisions.

Creating Symbols

You create a Symbol by calling the `Symbol()` function:

```
const sym1 = Symbol();
const sym2 = Symbol("description");

console.log(sym1 === sym2); // false - each Symbol is unique
```

- The optional string passed to `Symbol()` is a **description** for debugging purposes; it does not affect uniqueness.
- Symbols cannot be created with `new Symbol()` — they are not constructors.

Using Symbols as Object Property Keys

Because Symbols are unique, they are ideal for defining **hidden or non-colliding properties** on objects:

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

const user = {
  name: "Alice",
  [id]: 12345,  // Using Symbol as a key
};

console.log(user.name);  // Alice
console.log(user[id]);   // 12345
```

In this example:

- The property keyed by `id` cannot be accidentally overwritten by another string key `"id"`.
- Symbol-keyed properties are **not enumerated** in normal loops or `Object.keys()`, which helps avoid unintended exposure.

Why Use Symbols?

- **Avoid Property Name Collisions:** When multiple parts of code or libraries add properties, Symbols ensure keys won't clash.
- **Create Hidden/Internal Properties:** Symbol keys are not included in typical property enumerations.

- **Express Intent:** Using Symbols can indicate properties that have special meaning or usage.

15.1.1 Summary

Symbols are unique, immutable identifiers perfect for safe and collision-free object property keys. Their uniqueness and non-enumerability make them valuable tools in modern JavaScript for managing internal or special properties while avoiding interference in object structures.

15.2 Symbol Properties and Use Cases

Symbols serve as unique keys for object properties, offering distinct advantages over traditional string keys. When you use a Symbol as a property key, the property becomes **non-enumerable** by default, meaning it won't show up in standard loops or methods like `for...in` or `Object.keys()`. This feature is useful for adding metadata or internal details without interfering with normal object operations.

Symbol-Keyed Properties and Enumeration

Consider the following example:

```
const secret = Symbol("secret");

const obj = {
  name: "Bob",
  [secret]: "hiddenValue"
};

console.log(obj[secret]);  // "hiddenValue"

// Normal property enumeration
for (let key in obj) {
  console.log(key);  // Outputs: name
}

console.log(Object.keys(obj));  // ["name"]

// To get symbol properties, use Object.getOwnPropertySymbols()
const symbols = Object.getOwnPropertySymbols(obj);
console.log(symbols);           // [ Symbol(secret) ]
console.log(obj[symbols[0]]);   // "hiddenValue"
```

Here, the `secret` property is hidden from common enumeration techniques, making Symbols ideal for **private or internal data**.

Use Cases for Symbol Properties

- **Metadata:** Add extra info to objects without cluttering normal keys.
- **Private/Internal Properties:** Since Symbol keys are not included in many object operations, they help encapsulate data without true privacy (unlike private fields).
- **Constants:** Use Symbols as unique constants in your code to prevent accidental overwriting or name conflicts.

Example:

```
const STATUS = {
  READY: Symbol("ready"),
  RUNNING: Symbol("running"),
  DONE: Symbol("done")
};

function checkStatus(status) {
  if (status === STATUS.READY) {
    console.log("Ready to start");
  }
}
```

Using Symbols for constants guarantees unique values and safer comparisons.

15.2.1 Summary

Symbol properties are powerful tools for defining hidden or unique keys on objects. Their non-enumerability prevents accidental access and collision, making them ideal for metadata, private-like properties, and constants in JavaScript applications.

15.3 Well-Known Symbols

JavaScript includes a set of **well-known Symbols** that allow developers to customize and extend the language's built-in behaviors. These Symbols are used internally by JavaScript but can be overridden or implemented on your own objects to modify how they behave in certain operations.

Here are some of the most commonly used well-known Symbols:

`Symbol.iterator`

This Symbol lets you define **custom iteration behavior** for an object, enabling it to be used in `for...of` loops or with spread syntax (`...`).

```
const myIterable = {
  data: [10, 20, 30],
  [Symbol.iterator]() {
    let index = 0;
    const data = this.data;
    return {
      next() {
        if (index < data.length) {
          return { value: data[index++], done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for (const value of myIterable) {
  console.log(value);
}
// Output: 10, 20, 30
```

`Symbol.toStringTag`

This Symbol customizes the **default string description** of an object when using `Object.prototype.toString.call()`.

```
const obj = {
  [Symbol.toStringTag]: "CustomObject"
};

console.log(Object.prototype.toString.call(obj));
// Output: [object CustomObject]
```

Without this, the output would be `[object Object]`.

`Symbol.hasInstance`

This Symbol customizes the behavior of the `instanceof` operator. By defining this method, you control how an object determines whether another object is considered an instance.

```
class EvenNumberChecker {
  static [Symbol.hasInstance](value) {
    return typeof value === "number" && value % 2 === 0;
  }
}

console.log(2 instanceof EvenNumberChecker);  // true
console.log(3 instanceof EvenNumberChecker);  // false
```

15.3.1 Summary

Well-known Symbols like `Symbol.iterator`, `Symbol.toStringTag`, and `Symbol.hasInstance` provide powerful hooks to customize native language behavior. Implementing these Symbols on your objects enhances interoperability and allows you to define intuitive, expressive APIs that integrate seamlessly with JavaScript's built-in mechanisms.

Chapter 16.

Proxies and Reflect API

1. Proxy Syntax and Usage
2. Trap Handlers
3. Reflect API Overview

16 Proxies and Reflect API

16.1 Proxy Syntax and Usage

JavaScript **Proxies** are special objects that wrap another target object and allow you to intercept and customize fundamental operations like property access, assignment, method calls, and more. This powerful feature lets you add custom behavior such as validation, logging, or dynamic property handling without modifying the original object.

Basic Proxy Syntax

A Proxy is created using the constructor:

```
const proxy = new Proxy(target, handler);
```

- **`target`**: The original object you want to wrap.
- **`handler`**: An object containing *trap* functions that intercept operations on the target.

Example: Logging Property Access

Here's a simple example where a Proxy logs every property read from an object:

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

const loggedPerson = new Proxy(person, {
  get(target, property) {
    console.log(`Accessed property "${property}"`);
    return target[property];
  }
});

console.log(loggedPerson.name); // Logs: Accessed property "name" Output: Alice
console.log(loggedPerson.age);  // Logs: Accessed property "age" Output: 30
```

In this example, the `get` trap intercepts property reads. Whenever a property is accessed on `loggedPerson`, the trap logs the property name, then returns its value from the original `person`.

Example: Validating Property Assignments

Proxies can also intercept writes to properties. This example restricts assigning non-numeric values to an `age` property:

```
const user = { age: 25 };

const validatedUser = new Proxy(user, {
  set(target, property, value) {
    if (property === "age" && typeof value !== "number") {
      throw new TypeError("Age must be a number");
    }
    target[property] = value;
```

```
    return true; // indicates success
  }
});

validatedUser.age = 30;  // Works fine
console.log(validatedUser.age); // 30

// validatedUser.age = "old";  // Throws TypeError: Age must be a number
```

16.1.1 Summary

Proxies provide a flexible way to customize how objects behave by intercepting their fundamental operations. With just a few lines of code, you can add logging, validation, or even create entirely new behaviors without touching the original object. This makes Proxies a powerful tool for advanced JavaScript patterns such as data binding, validation frameworks, and security wrappers.

16.2 Trap Handlers

In JavaScript Proxies, **trap handlers** are special methods defined on the handler object that intercept and customize fundamental operations performed on the target object. These traps give you fine-grained control over behaviors like property access, assignment, existence checks, deletion, and more.

How Trap Handlers Work

When you perform an operation on a Proxy, the corresponding trap method (if defined) is invoked instead of the default behavior. Each trap receives parameters that let you inspect or modify the operation, and you can choose whether to forward the operation to the target or override it completely.

Common Trap Handlers

Here are some frequently used traps and their purposes:

Trap Name	Description
`get`	Intercepts property access (`proxy.prop`)
`set`	Intercepts property assignment (`proxy.prop = val`)
`has`	Intercepts the `in` operator (`prop in proxy`)
`deleteProperty`	Intercepts `delete proxy.prop`
`ownKeys`	Intercepts operations returning property keys (like `Object.keys()`)

Example: Using `get` and `set` for Validation and Logging

```
const user = { name: "Jane", age: 28 };

const proxy = new Proxy(user, {
  get(target, prop) {
    console.log(`Getting property: ${prop}`);
    return Reflect.get(target, prop); // Forward to target
  },
  set(target, prop, value) {
    if (prop === "age" && typeof value !== "number") {
      throw new TypeError("Age must be a number");
    }
    console.log(`Setting property ${prop} to ${value}`);
    return Reflect.set(target, prop, value); // Forward to target
  }
});

console.log(proxy.name);  // Logs "Getting property: name" and outputs "Jane"
proxy.age = 30;           // Logs "Setting property age to 30"
proxy.age = "old";        // Throws TypeError
```

Example: Customizing `has` and `deleteProperty`

```
const data = { secret: "hidden", visible: true };

const handler = {
  has(target, prop) {
    if (prop === "secret") {
      return false; // Hide 'secret' property from `in`
    }
    return prop in target;
  },
  deleteProperty(target, prop) {
    console.log(`Deleting property: ${prop}`);
    return Reflect.deleteProperty(target, prop);
  }
};

const proxyData = new Proxy(data, handler);

console.log("secret" in proxyData);  // false
console.log("visible" in proxyData); // true

delete proxyData.visible; // Logs "Deleting property: visible"
console.log(proxyData.visible); // undefined
```

Summary

Trap handlers are at the core of Proxies' power, enabling you to **customize behavior** for almost any fundamental object operation. By implementing traps like `get`, `set`, `has`, and `deleteProperty`, you can build powerful abstractions such as validation layers, access logs, virtual properties, or security wrappers — all without changing the original object's structure. Using the `Reflect` API inside traps is recommended to forward operations to the target

cleanly and consistently.

16.3 Reflect API Overview

The **Reflect API** is a built-in JavaScript object introduced in ES6 that provides methods corresponding to many of the language's internal operations. It acts as a standardized way to perform common low-level tasks like property access, assignment, deletion, and more.

Why Use Reflect in Proxy Handlers?

When writing **Proxy trap handlers**, you often want to customize behavior but still delegate to the original operation to maintain the expected object behavior. The Reflect API methods make this easier by providing consistent, reliable ways to perform default operations.

Without Reflect, you'd have to manually perform actions like getting or setting properties, which can be error-prone and verbose. Reflect methods return boolean values or the property value, helping Proxy traps handle success or failure properly.

Key Reflect Methods

Here are some commonly used Reflect methods in Proxy traps:

- `Reflect.get(target, property, receiver)` Retrieves the property value from the target object. The `receiver` helps maintain correct `this` context, especially with getters.
- `Reflect.set(target, property, value, receiver)` Sets a property value on the target. Returns `true` if successful.
- `Reflect.deleteProperty(target, property)` Deletes a property from the target. Returns `true` if the deletion succeeds.
- `Reflect.has(target, property)` Checks if a property exists on the target (used in `in` operator).

Example: Using Reflect in Proxy Traps

```
const person = { name: "Alex", age: 25 };

const proxy = new Proxy(person, {
  get(target, prop, receiver) {
    console.log(`Accessing ${prop}`);
    return Reflect.get(target, prop, receiver);
  },
  set(target, prop, value, receiver) {
    console.log(`Setting ${prop} to ${value}`);
    return Reflect.set(target, prop, value, receiver);
  },
  deleteProperty(target, prop) {
```

```
    console.log(`Deleting property: ${prop}`);
    return Reflect.deleteProperty(target, prop);
  }
});

console.log(proxy.name);   // Logs "Accessing name" then "Alex"
proxy.age = 30;            // Logs "Setting age to 30"
delete proxy.name;         // Logs "Deleting property: name"
```

Summary

The Reflect API helps make Proxy handlers **transparent and safe** by allowing traps to forward operations to the original target cleanly. Using Reflect ensures that your proxies behave as expected, preserving JavaScript's internal semantics while enabling powerful customization.

Chapter 17.

Working with JSON

1. JSON Syntax and Parsing
2. Converting Objects to JSON
3. Practical JSON Use Cases

17 Working with JSON

17.1 JSON Syntax and Parsing

JSON (JavaScript Object Notation) is a lightweight, text-based data interchange format widely used for exchanging data between servers and clients. It's easy for humans to read and write, and easy for machines to parse and generate.

JSON Syntax Rules

JSON syntax is a subset of JavaScript object syntax with strict rules:

- **Data types allowed**:
  - Objects (curly braces `{}` with key-value pairs)
  - Arrays (square brackets `[]`)
  - Strings (double-quoted `"text"`)
  - Numbers (integer or floating-point)
  - Booleans (`true` or `false`)
  - `null`
- **Keys in objects must be strings** enclosed in double quotes.
- **No functions, `undefined`, or comments** are allowed in JSON.
- Whitespace (spaces, tabs, line breaks) is ignored and used for readability.

Parsing JSON Strings

To convert a JSON string into a JavaScript object, use the built-in method:

```
const jsonString = '{"name":"Alice","age":30,"isStudent":false}';
const obj = JSON.parse(jsonString);
console.log(obj.name);  // Output: Alice
console.log(obj.age);   // Output: 30
```

If the JSON string is invalid, `JSON.parse()` throws a **SyntaxError**:

```
const invalidJson = '{name:"Alice", age:30}'; // Keys not in double quotes

try {
  JSON.parse(invalidJson);
} catch (error) {
  console.error("Parsing failed:", error.message);
}
// Output: Parsing failed: Unexpected token n in JSON at position 1
```

Example of Array Parsing

```
const jsonArray = '["apple", "banana", "cherry"]';
const fruits = JSON.parse(jsonArray);
```

```
console.log(fruits[1]);  // Output: banana
```

Summary

JSON is a universal data format with simple syntax rules designed for easy data exchange. The `JSON.parse()` method safely converts JSON text into JavaScript objects, but be mindful of strict syntax requirements to avoid parsing errors.

17.2 Converting Objects to JSON

In JavaScript, you can convert objects and arrays into JSON strings using the built-in method `JSON.stringify()`. This process is essential when sending data to APIs or saving data in a text format.

Basic Usage

`JSON.stringify()` takes a JavaScript value and returns a JSON-formatted string.

```
const obj = { name: "Alice", age: 30, hobbies: ["reading", "hiking"] };
const jsonString = JSON.stringify(obj);
console.log(jsonString);
// Output: {"name":"Alice","age":30,"hobbies":["reading","hiking"]}
```

Formatting Output with Spacing

To make the JSON string more readable, especially for logging or configuration files, `JSON.stringify()` accepts a third argument called `space`. It controls indentation and line breaks.

```
const obj = { name: "Alice", age: 30, hobbies: ["reading", "hiking"] };
const prettyJson = JSON.stringify(obj, null, 2);
console.log(prettyJson);

/* Output:
{
  "name": "Alice",
  "age": 30,
  "hobbies": [
    "reading",
    "hiking"
  ]
}
*/
```

Using Replacer Functions or Arrays

The second parameter of `JSON.stringify()` can be:

- **A replacer function**: allows selective transformation of key-value pairs during stringi-

fication.

- **An array of keys**: specifies which properties should be included.

Example with replacer function:

```
const user = { name: "Bob", password: "secret", age: 25 };
const safeJson = JSON.stringify(user, (key, value) => {
  if (key === "password") return undefined; // Exclude password
  return value;
});
console.log(safeJson);
// Output: {"name":"Bob","age":25}
```

Example with array replacer:

```
const user = { name: "Bob", password: "secret", age: 25 };
const filteredJson = JSON.stringify(user, ["name", "age"]);
console.log(filteredJson);
// Output: {"name":"Bob","age":25}
```

Handling Special Values

- **Functions and `undefined`** are omitted when they appear as object properties.
- **In arrays, `undefined` or functions are serialized as `null`.**

Example:

```
const data = {
  name: "Carol",
  greet: () => "Hi!",
  age: undefined,
  scores: [10, undefined, 30]
};
console.log(JSON.stringify(data));
// Output: {"name":"Carol","scores":[10,null,30]}
```

Summary

`JSON.stringify()` converts JavaScript data into JSON strings with options to customize output using replacers and spacing. It gracefully handles special cases, making it a versatile tool for data serialization.

17.3 Practical JSON Use Cases

JSON is the go-to format for exchanging data between web applications, servers, and clients due to its simplicity and compatibility. Here are some common real-world scenarios where JSON plays a crucial role.

Fetching JSON Data from APIs

Modern web applications often retrieve data from RESTful APIs that return JSON. Using the `fetch()` API, you can request JSON data and parse it easily.

```
fetch('https://api.example.com/users')
  .then(response => response.json()) // Parse JSON response
  .then(data => {
    console.log(data); // Use the JavaScript object/array
  })
  .catch(error => console.error('Error fetching JSON:', error));
```

The `.json()` method on the response reads the JSON stream and converts it into a usable JavaScript object.

Storing Data in Browser Storage

Browsers provide storage options like `localStorage` and `sessionStorage` which store only strings. JSON lets you save complex data structures by converting them to strings.

```
const settings = { theme: 'dark', notifications: true };
localStorage.setItem('userSettings', JSON.stringify(settings));

// Later, retrieve and parse:
const storedSettings = JSON.parse(localStorage.getItem('userSettings'));
console.log(storedSettings.theme); // Output: dark
```

Always remember to check for `null` when retrieving data that might not exist.

Sending JSON in HTTP Requests

When sending data to a server, such as submitting a form, you typically send JSON in the request body with the correct headers.

```
const newUser = { name: 'Dana', email: 'dana@example.com' };

fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(newUser)
})
  .then(response => response.json())
  .then(data => console.log('User created:', data))
  .catch(error => console.error('Error:', error));
```

Common Pitfalls

- **Parsing errors:** Always handle errors when calling `JSON.parse()` on external data to avoid crashes.
- **Circular references:** Objects with circular references cannot be stringified with

`JSON.stringify()`.

- **Data type loss:** JSON only supports a limited set of data types; functions, `undefined`, and special objects like `Date` are not preserved and require special handling.

Summary

JSON is indispensable for client-server communication, client-side storage, and configuration. Familiarity with its practical uses and limitations helps build robust, maintainable JavaScript applications.