JavaScript Object-Oriented Programming

JavaScript Object-Oriented Programming

From Basics to Advanced Design

readbytes.github.io

2025-07-12

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to JavaScript and OOP Concepts

1. What is JavaScript and Object-Oriented Programming
2. Overview of JavaScript's OOP Model
3. Your First JavaScript Object: A Simple Example

1 Introduction to JavaScript and OOP Concepts

1.1 What is JavaScript and Object-Oriented Programming

JavaScript is a versatile, high-level programming language that has become a cornerstone of modern web development. Originally created in 1995 by Brendan Eich at Netscape, JavaScript was designed to add interactivity and dynamic behavior to otherwise static web pages. Over the years, it has evolved from a simple scripting language into a powerful, multi-paradigm language capable of building complex applications both in the browser and on servers.

Object-Oriented Programming, often called OOP, is a powerful programming paradigm that helps developers organize and structure their code in a way that models the real world more naturally. Instead of thinking purely in terms of actions and procedures, OOP focuses on *objects*—entities that represent things, concepts, or components with their own properties and behaviors.

At its core, OOP is built on several key principles:

- **Objects:** Think of objects as real-world things like a car, a person, or a bank account. Each object has *properties* (attributes describing it, such as color or balance) and *methods* (actions it can perform, like driving or depositing money). In programming, objects bundle together data and behavior, making code easier to understand and maintain.
- **Classes:** Classes are like blueprints or templates for creating objects. Imagine a blueprint for a house—you can use it to build many houses with the same design but different details. Similarly, a class defines the structure and capabilities an object will have. Objects created from the same class are called instances.
- **Inheritance:** This principle allows one class to *inherit* properties and methods from another, promoting code reuse and hierarchy. For example, consider animals: a "Dog" class can inherit general traits from an "Animal" class but also have its own specific behaviors, like barking. Inheritance models relationships and shared behavior efficiently.
- **Encapsulation:** Encapsulation means hiding the internal details of an object and exposing only what is necessary. Think of it as a TV remote: you interact with buttons without needing to understand the complex electronics inside. In programming, encapsulation protects an object's data and ensures it can only be modified in controlled ways, improving security and reducing errors.
- **Polymorphism:** This fancy term means "many forms" and allows objects of different classes to be treated through a common interface. For instance, if both "Cat" and "Dog" classes have a method called `makeSound()`, you can call `makeSound()` on any animal without worrying about the specific type. Polymorphism simplifies code by enabling flexible and interchangeable object behavior.

1.1.1 OOP in JavaScript

Object-Oriented Programming (OOP) brings significant advantages to JavaScript development by helping organize code into reusable, modular, and maintainable pieces. As JavaScript projects grow larger and more complex, these benefits become crucial for building robust applications.

One of the biggest strengths of OOP is **reusability**. By defining objects and classes that encapsulate data and behavior, developers can create reusable components. Instead of rewriting similar code multiple times, you can instantiate new objects or extend existing classes, saving time and reducing errors.

OOP also encourages **modularity**. By breaking an application into distinct objects, each with its own responsibilities, the codebase becomes easier to understand and manage. This separation of concerns allows developers to focus on individual parts without being overwhelmed by the whole system, improving clarity and reducing bugs.

1.2 Overview of JavaScript's OOP Model

JavaScript takes a unique approach to object-oriented programming compared to many other popular languages. Instead of using classical class-based inheritance, JavaScript is built around a **prototype-based inheritance** model. This fundamental difference shapes how objects are created, extended, and linked in JavaScript.

In classical OOP languages like Java or C++, classes serve as blueprints for creating objects, and inheritance happens between classes in a strict hierarchy. JavaScript, however, does not require classes to create objects or establish inheritance relationships. Instead, each object has an internal link to another object called its **prototype**. When you try to access a property or method on an object, JavaScript first looks on that object itself. If it doesn't find it there, it follows the prototype link to search on the prototype object. This chain of linked prototypes continues up until it reaches an object with a `null` prototype, called the root.

This **prototype chain** mechanism allows objects to inherit properties and methods directly from other objects, making the inheritance model more flexible and dynamic. You can modify an object's prototype at runtime to add or change behavior, which is quite different from the static inheritance structures in classical languages.

Because of this prototype-based design, JavaScript's OOP can be more lightweight and adaptable. However, it also means that developers coming from classical OOP backgrounds need to understand this difference in how inheritance and object relationships work under the hood.

Recognizing that classical syntax is often more intuitive, ES6 introduced **classes** to JavaScript in 2015. However, these classes are essentially **syntactic sugar** — a cleaner, more familiar way to write the same prototype-based inheritance code behind the scenes. ES6 classes

allow developers to define constructors, methods, and inheritance hierarchies in a style that resembles traditional OOP languages, making the language more approachable while maintaining its unique prototype model.

1.3 Your First JavaScript Object: A Simple Example

Let's start by creating a very simple JavaScript object that represents a **person**. This example will help you understand how objects in JavaScript bundle together **data** (properties) and **behavior** (methods).

Here's a basic object literal:

```
const person = {
  firstName: "Alice",
  lastName: "Johnson",
  age: 30,

  // Method to return the full name
  getFullName: function() {
    return this.firstName + " " + this.lastName;
  },

  // Method to greet
  greet: function() {
    console.log("Hello, my name is " + this.getFullName());
  }
};
```

1.3.1 Explanation, line by line:

- `const person = { ... };` This line creates a new object and assigns it to the variable `person`. The curly braces `{}` define an **object literal**, which is a simple way to create an object in JavaScript.
- `firstName: "Alice",` This is a **property** of the object named `firstName`, holding the string `"Alice"`. Properties are like variables attached to the object.
- `lastName: "Johnson",` Another property storing the last name.
- `age: 30,` A numeric property representing the person's age.
- `getFullName: function() { ... },` This defines a **method** — a function attached to the object. The method `getFullName` returns the full name by combining `firstName` and `lastName`. Notice the use of `this` to refer to the current object, allowing access to its properties.

- `greet: function() { ... }` Another method named `greet` that prints a greeting message to the console. It calls `this.getFullName()` to get the person's full name dynamically.

1.3.2 Using the object:

```
console.log(person.firstName); // Output: Alice

console.log(person.getFullName()); // Output: Alice Johnson

person.greet(); // Output: Hello, my name is Alice Johnson
```

- `person.firstName` accesses the `firstName` property of the `person` object.
- `person.getFullName()` calls the method to get the full name.
- `person.greet()` calls the greeting method, which uses another method inside the object.

Chapter 2.

JavaScript Objects Basics

1. Creating Objects Using Object Literals
2. Accessing and Modifying Object Properties
3. Methods in Objects
4. The `this` Keyword in JavaScript Objects
5. Practical: Creating a Simple Contact Card Object

2 JavaScript Objects Basics

2.1 Creating Objects Using Object Literals

In JavaScript, one of the simplest and most common ways to create objects is by using **object literals**. An object literal is a comma-separated list of key-value pairs wrapped inside curly braces `{}`. This syntax is concise, easy to read, and perfect for grouping related data and behavior in a small to medium-sized structure.

2.1.1 Basic Syntax

Here is a simple example of an object literal:

```
const book = {
  title: "JavaScript Essentials",
  author: "Jane Smith",
  yearPublished: 2022
};
```

- The object `book` has three **properties**: `title`, `author`, and `yearPublished`.
- Each property consists of a **key** (like `title`) and a **value** (like `"JavaScript Essentials"`).

2.1.2 Adding Methods

Object literals can also include **methods**, which are functions stored as object properties:

```
const book = {
  title: "JavaScript Essentials",
  author: "Jane Smith",
  yearPublished: 2022,
  getSummary: function() {
    return `${this.title} by ${this.author}, published in ${this.yearPublished}`;
  }
};

console.log(book.getSummary());
// Output: JavaScript Essentials by Jane Smith, published in 2022
```

- The `getSummary` method returns a string summarizing the book's details.
- Inside the method, `this` refers to the `book` object itself.

2.1.3 Nested Objects and Arrays as Properties

Object literals can hold complex data by nesting objects or arrays:

```
const person = {
  name: "John Doe",
  age: 28,
  address: {
    street: "123 Main St",
    city: "Anytown",
    country: "USA"
  },
  hobbies: ["reading", "hiking", "gaming"]
};
```

- `address` is a **nested object** representing a detailed property of the person.
- `hobbies` is an **array** containing multiple string values.

You can access nested properties using dot notation:

```
console.log(person.address.city);  // Output: Anytown
console.log(person.hobbies[1]);    // Output: hiking
```

2.1.4 Advantages of Object Literals

- **Simplicity:** Object literals provide a straightforward way to define objects without needing explicit constructors or classes.
- **Readability:** The structure closely resembles JSON, making it intuitive to write and understand.
- **Flexibility:** Properties can store any data type, including other objects, arrays, functions, or primitive values.
- **Convenience:** Ideal for representing small to medium collections of related data or behavior, such as configuration settings, user profiles, or UI components.

2.2 Accessing and Modifying Object Properties

In JavaScript, objects are collections of key-value pairs, and interacting with these properties is fundamental. You can **access**, **modify**, and even **add** properties dynamically using two main syntaxes: **dot notation** and **bracket notation**. Understanding when and how to use each is essential for effective JavaScript programming.

2.2.1 Accessing Properties

Dot Notation

The most common way to access properties is using **dot notation**:

```
const car = {
  make: "Toyota",
  model: "Camry",
  year: 2020
};

console.log(car.make);   // Output: Toyota
console.log(car.year);   // Output: 2020
```

- Dot notation is simple and clean.
- It works well when property names are valid JavaScript identifiers (no spaces or special characters).

Bracket Notation

Bracket notation uses square brackets `[]` with a string or variable representing the property name:

```
console.log(car["model"]);  // Output: Camry
```

Bracket notation is especially useful when:

- Property names contain **spaces** or **special characters**:

```
const person = {
  "first name": "Alice",
  "favorite-color": "blue"
};

console.log(person["first name"]);       // Output: Alice
console.log(person["favorite-color"]);   // Output: blue
```

- Property names are **dynamic**, stored in variables:

```
const propName = "make";
console.log(car[propName]);   // Output: Toyota
```

2.2.2 Modifying Properties

You can change the value of an existing property using either notation:

```
car.year = 2021;
console.log(car.year);  // Output: 2021
```

```
car["model"] = "Corolla";
console.log(car.model); // Output: Corolla
```

2.2.3 Adding New Properties

Objects in JavaScript are dynamic, so you can add new properties anytime:

```
car.color = "red";
console.log(car.color);  // Output: red

car["owner"] = "John Doe";
console.log(car.owner);  // Output: John Doe
```

2.2.4 Handling Undefined Properties

If you try to access a property that does not exist, JavaScript returns `undefined` rather than throwing an error:

```
console.log(car.owner);  // Output: John Doe
console.log(car.price);  // Output: undefined
```

This behavior allows you to check whether a property exists before using it:

```
if (car.price === undefined) {
  console.log("Price is not set.");
}
```

2.2.5 Summary

- Use **dot notation** for simple, known property names without spaces or special characters.
- Use **bracket notation** when property names are dynamic, contain spaces, or special characters.
- You can modify existing properties or add new ones at any time using either notation.
- Accessing nonexistent properties returns `undefined`, which you can check to avoid errors.

Mastering these ways to access and modify object properties helps you interact flexibly and safely with JavaScript objects in all kinds of scenarios.

2.3 Methods in Objects

In JavaScript, **functions** can be stored inside objects as properties. When a function is associated with an object like this, it's called a **method**. Methods allow objects not only to hold data but also to perform actions related to that data, making objects more dynamic and powerful.

2.3.1 Defining Methods in Objects

Here's an example of an object with a method:

```
const user = {
  name: "Emma",
  age: 25,

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

user.greet(); // Output: Hello, my name is Emma
```

- The `greet` property is a function attached to the `user` object, making it a method.
- Inside the method, we use `this.name` to refer to the `name` property of the current object (`user`).
- Calling `user.greet()` executes the method, which accesses the object's own data to display a personalized greeting.

2.3.2 Why Use `this`?

The keyword `this` is crucial in methods because it lets the method refer to the **current object** it belongs to, no matter what the object's name is. This allows the method to dynamically access or modify the object's properties:

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

  area: function() {
    return this.width * this.height;
  }
};

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

Here, the `area` method calculates the rectangle's area by multiplying its own `width` and `height` properties using `this`.

2.3.3 ES6 Method Shorthand

With ES6, JavaScript introduced a cleaner and shorter syntax for defining methods inside object literals:

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

user.greet(); // Output: Hi, I'm Emma!
```

- Instead of writing `greet: function() { ... }`, you can simply write `greet() { ... }`.
- This shorthand improves readability and is widely used in modern JavaScript code.

2.3.4 Summary

- Methods are functions stored as properties inside objects.
- They use the `this` keyword to access and manipulate the object's own properties.
- Methods let objects bundle **data and behavior** together, following the core principles of object-oriented programming.
- ES6 introduced a concise method definition syntax that makes code cleaner and easier to read.

By using methods, you enable your objects to perform actions related to their data, making your JavaScript code more expressive and organized.

2.4 The `this` Keyword in JavaScript Objects

The keyword `this` is one of the most important—and sometimes confusing—aspects of JavaScript. It refers to the **context** in which a function is executed, and understanding how `this` behaves is crucial when working with objects and their methods.

2.4.1 What Does `this` Refer To?

In JavaScript, the value of `this` depends on **how** a function is called, not where it is defined. This means `this` can point to different things depending on the **invocation pattern**.

2.4.2 `this` Inside Object Methods

When a function is called as a method of an object, `this` refers to the object itself:

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

person.greet();  // Output: Hello, my name is Liam
```

Here, `this.name` accesses the `name` property of the `person` object because `greet` was invoked as a method of `person`.

2.4.3 `this` in the Global Context

When a function is called without being attached to an object (i.e., in the global context), `this` refers to the global object. In browsers, this is typically `window`:

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

sayHello();  // Output: [object Window] (in browsers)
```

In **strict mode**, `this` inside a standalone function defaults to `undefined` instead of the global object, helping prevent accidental bugs.

2.4.4 `this` Inside Nested Functions

A common pitfall occurs when using nested functions inside methods. Inside a nested function, `this` does **not** automatically inherit the outer method's `this`:

```
const user = {
  name: "Mia",
```

```
  greet: function() {
    function inner() {
      console.log(this.name);
    }
    inner();  // Output: undefined (or error in strict mode)
  }
};

user.greet();
```

Here, `inner()` is a normal function call, so `this` refers to the global object or `undefined` in strict mode—not the `user` object. As a result, `this.name` is `undefined`.

2.4.5 How to Fix `this` in Nested Functions

There are several ways to preserve the correct `this` inside nested functions:

1. **Assign `this` to a variable**

```
const user = {
  name: "Mia",
  greet: function() {
    const self = this;  // save reference to outer this
    function inner() {
      console.log(self.name);
    }
    inner();  // Output: Mia
  }
};

user.greet();
```

2. **Use arrow functions**

Arrow functions do not have their own `this` and inherit it from the surrounding context:

```
const user = {
  name: "Mia",
  greet: function() {
    const inner = () => {
      console.log(this.name);
    };
    inner();  // Output: Mia
  }
};

user.greet();
```

2.4.6 Summary of Common `this` Pitfalls

- Calling a method on an object sets `this` to the object.
- Calling a standalone function sets `this` to the global object (`window`) or `undefined` in strict mode.
- Nested regular functions lose the outer `this` unless handled explicitly.
- Arrow functions inherit `this` from their surrounding scope.
- Using variables like `self` or arrow functions are common patterns to maintain correct `this` inside nested functions.

Understanding how `this` works helps you avoid bugs related to context and write clearer, more predictable object-oriented JavaScript code.

2.5 Practical: Creating a Simple Contact Card Object

Let's put what you've learned into practice by building a **Contact Card** object. This example will help reinforce your understanding of object literals, properties, methods, and the use of `this`.

We'll create an object that represents a person's contact information, with the ability to display and update that information.

2.5.1 Step-by-Step: Contact Card Object

```
const contactCard = {
  // Properties
  name: "Olivia Rivera",
  phone: "555-1234",
  email: "olivia@example.com",

  // Method to display contact info
  displayInfo() {
    console.log("Contact Info:");
    console.log("Name: " + this.name);
    console.log("Phone: " + this.phone);
    console.log("Email: " + this.email);
  },

  // Method to update phone number
  updatePhone(newPhone) {
    this.phone = newPhone;
    console.log("Phone number updated to: " + this.phone);
  },

  // Method to update email address
```

```
  updateEmail(newEmail) {
    this.email = newEmail;
    console.log("Email updated to: " + this.email);
  }
};

// Display original contact info
contactCard.displayInfo();

/* Output:
Contact Info:
Name: Olivia Rivera
Phone: 555-1234
Email: olivia@example.com
*/

// Update phone and email
contactCard.updatePhone("555-6789");
contactCard.updateEmail("olivia.r@example.com");

// Display updated contact info
contactCard.displayInfo();

/* Output:
Contact Info:
Name: Olivia Rivera
Phone: 555-6789
Email: olivia.r@example.com
*/
```

2.5.2 Key Concepts in the Example

- **Properties** (`name`, `phone`, `email`) store the contact's basic information.
- **Methods** (`displayInfo`, `updatePhone`, `updateEmail`) perform actions on the object's data.
- The use of `this` ensures each method refers to the current instance of the object, making the object **self-contained and reusable**.

2.5.3 Best Practices Demonstrated

- **Descriptive naming**: Clear property and method names like `displayInfo` and `updateEmail` make the object easy to understand.
- **Encapsulation**: The contact's data and the behavior to manage that data are grouped within a single object.
- **Maintainability**: If the internal structure of the object changes (e.g., new fields are added), you only need to update methods inside the object, not elsewhere in the

program.

2.5.4 Try It Yourself

- Add new properties like `address` or `birthday`.
- Add a `getSummary()` method that returns a string with all the contact details.
- Create multiple contact objects using the same pattern.

By building this simple contact card, you've seen how object literals let you organize data and behavior in a clean, structured way. This is a foundational skill in JavaScript and sets the stage for more advanced object-oriented programming concepts.

Chapter 3.

Prototypes and Inheritance Fundamentals

1. What is a Prototype?
2. Prototype Chain Explained
3. Adding Properties and Methods to Prototypes
4. Inheriting from Prototypes
5. Practical: Prototype-Based Inheritance Example

3 Prototypes and Inheritance Fundamentals

3.1 What is a Prototype?

In JavaScript, **prototypes** are a fundamental concept in how objects work and inherit behavior. At a basic level, a prototype is just another object that serves as a **template** from which other objects can inherit properties and methods.

Think of a prototype like a **recipe card** in a kitchen. If you create multiple dishes using the same recipe, each dish is different, but they all share the same underlying instructions. In JavaScript, when you create an object, it can automatically follow a "recipe" from another object—its prototype.

3.1.1 Every Object Has a Prototype

Behind the scenes, **every JavaScript object** has an internal link to another object called its **prototype**. This prototype object acts as a shared source of properties and methods. If you try to access a property or method on an object and it doesn't exist there, JavaScript automatically looks for it in the object's prototype.

Here's a simple example:

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

const student = Object.create(person);

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

- The `student` object was created using `Object.create(person)`, which means `person` becomes the prototype of `student`.
- Even though `greet` is not directly defined on `student`, JavaScript finds it in `person` (its prototype) and runs it.

3.1.2 Prototype as a Shared Template

Using prototypes helps JavaScript objects share behavior **without duplicating code**. Instead of giving every object its own copy of a method, we can define the method once on a prototype, and let multiple objects use it.

This model is different from many other languages (like Java or C++) that use **class-based**

inheritance, where classes are the primary way to define templates for objects. In JavaScript, **prototypes are the original way** to achieve inheritance—though newer syntax (like `class`) still relies on prototypes behind the scenes.

3.1.3 Summary

- A **prototype** is an object from which other objects inherit properties and methods.
- Every JavaScript object has an internal reference to a prototype object.
- If a property or method isn't found directly on an object, JavaScript looks for it on the prototype.
- Prototypes provide a powerful and memory-efficient way to share behavior between objects.

Understanding prototypes gives you insight into how JavaScript truly handles inheritance and sets the foundation for deeper object-oriented techniques.

3.2 Prototype Chain Explained

In JavaScript, inheritance is made possible through a mechanism called the **prototype chain**. This chain defines how objects can inherit properties and methods from other objects.

3.2.1 What Is the Prototype Chain?

When you access a property or method on an object, JavaScript first looks for that property directly **on the object itself**. If it doesn't find it there, it follows an internal link to the object's **prototype** and searches there. If the property still isn't found, the search continues up the chain until it reaches the end—usually an object called `Object.prototype`, which is the root of most JavaScript objects.

If the property is not found anywhere along the chain, the result is `undefined`.

3.2.2 Step-by-Step Example

Let's walk through an example that shows how this lookup works:

```
function Animal() {
  this.type = "animal";
```

```
}

Animal.prototype.sound = function() {
  return "Some generic sound";
};

const dog = new Animal();

console.log(dog.type);      // "animal" - found on the instance itself
console.log(dog.sound());  // "Some generic sound" - found on Animal.prototype
console.log(dog.toString()); // Found on Object.prototype
```

Property Lookup Path:

1. Look for `sound` on `dog` → not found.
2. Look on `dog.__proto__` (which is `Animal.prototype`) → found.
3. Look for `toString` on `dog` → not found.
4. Look on `Animal.prototype` → not found.
5. Look on `Object.prototype` → found.
6. If not found here, return `undefined`.

3.2.3 Visualizing the Prototype Chain

`dog → Animal.prototype → Object.prototype → null`

Each arrow (→) represents the internal prototype link (`[[Prototype]]`).

In most environments, this internal link can be accessed with the special `__proto__` property (though it's better to use `Object.getPrototypeOf()`):

```
console.log(Object.getPrototypeOf(dog) === Animal.prototype); // true
console.log(Object.getPrototypeOf(Animal.prototype) === Object.prototype); // true
```

3.2.4 Why the Prototype Chain Matters

- It **enables inheritance**: Methods and properties shared across many objects can be placed on a prototype, allowing all instances to reuse them.
- It **saves memory**: Shared methods aren't duplicated for each object.
- It **provides fallback behavior**: Objects can delegate behavior to their prototypes if something isn't defined directly on them.

3.2.5 Summary

The prototype chain is JavaScript's way of handling inheritance. When accessing a property, JavaScript checks the object itself, then follows the chain of prototypes until it finds the property or reaches the end. This elegant system makes inheritance flexible and efficient—and understanding it is key to mastering JavaScript OOP.

3.3 Adding Properties and Methods to Prototypes

One of the most powerful features of JavaScript's object-oriented model is the ability to **add shared properties and methods to a constructor function's prototype**. This approach ensures that **all instances** created from that constructor can access the same functionality without duplicating it in memory.

3.3.1 Why Use Prototypes?

By default, properties defined inside a constructor function (using `this`) are created for **each individual instance**, which can lead to redundant memory usage. In contrast, methods and shared values placed on the prototype are **shared across all instances**, making your code more **memory-efficient** and easier to maintain.

3.3.2 Basic Example: Adding to a Prototype

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

// Add method to the prototype
Person.prototype.greet = function() {
  console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
};

const alice = new Person("Alice", 30);
const bob = new Person("Bob", 25);

alice.greet();  // Output: Hi, I'm Alice and I'm 30 years old.
bob.greet();    // Output: Hi, I'm Bob and I'm 25 years old.
```

- The `greet` method is **not copied** to each instance.
- Instead, both `alice` and `bob` **share** the same `greet` function via the prototype.

3.3.3 Adding Properties After Instances Are Created

You can modify the prototype **even after** instances are created, and all instances will have access to the new methods or properties:

```
// Add a new method later
Person.prototype.sayGoodbye = function() {
  console.log(`${this.name} says goodbye!`);
};

alice.sayGoodbye();  // Output: Alice says goodbye!
bob.sayGoodbye();    // Output: Bob says goodbye!
```

This dynamic behavior is possible because JavaScript looks up the prototype **at the time the property or method is accessed**, not when the object is created.

3.3.4 Memory Efficiency Benefits

If you were to define methods inside the constructor, each instance would get its **own copy**:

```
function Person(name) {
  this.name = name;
  this.greet = function() {
    console.log("Hello, I'm " + this.name);
  };
}

const a = new Person("Anna");
const b = new Person("Ben");

console.log(a.greet === b.greet);  // false (two different functions)
```

Compare this to using the prototype:

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

Person.prototype.greet = function() {
  console.log("Hello, I'm " + this.name);
};

const a = new Person("Anna");
const b = new Person("Ben");

console.log(a.greet === b.greet);  // true (shared function)
```

Sharing methods via the prototype avoids creating multiple copies in memory—especially important when creating many instances.

3.3.5 Summary

- Adding properties or methods to a constructor's prototype lets **all instances share** them.
- This practice improves **memory efficiency** and **performance**.
- You can modify prototypes at any time, and changes are reflected across all existing and future instances.
- Prototype-based sharing is a fundamental part of JavaScript's object system and a cornerstone of effective object-oriented design in the language.

3.4 Inheriting from Prototypes

In JavaScript, **inheritance** allows one object to reuse the properties and methods of another. Before the introduction of ES6 classes, JavaScript implemented inheritance using **constructor functions** and their associated **prototypes**. This system is known as **prototype-based inheritance**.

3.4.1 How Inheritance Works

Every function in JavaScript has a `prototype` property. When you use a constructor function to create an object, that object gets linked to the constructor's prototype. To set up inheritance, you link one constructor's prototype to an instance (or copy) of another constructor's prototype.

3.4.2 Example: Inheriting from a Parent Constructor

Let's walk through an example where a `Dog` constructor inherits from an `Animal` constructor:

```
// Parent constructor
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  console.log(`${this.name} makes a noise.`);
};

// Child constructor
function Dog(name, breed) {
  Animal.call(this, name); // Call parent constructor
  this.breed = breed;
```

```
}

// Set up prototype inheritance
Dog.prototype = Object.create(Animal.prototype);

// Correct the constructor reference
Dog.prototype.constructor = Dog;

// Add method specific to Dog
Dog.prototype.bark = function() {
  console.log(`${this.name} barks!`);
};

// Create a Dog instance
const myDog = new Dog("Rex", "German Shepherd");

myDog.speak(); // Output: Rex makes a noise.
myDog.bark();  // Output: Rex barks!
```

3.4.3 Step-by-Step Explanation

1. **Animal Constructor** Defines a shared property `name` and a method `speak()` on its prototype.
2. **Dog Constructor** Calls the `Animal` constructor using `Animal.call(this, name)` to initialize `name` on the child instance.
3. **Inheritance Setup**
   - `Dog.prototype = Object.create(Animal.prototype)` This sets `Dog.prototype` to a new object that inherits from `Animal.prototype`.
   - `Dog.prototype.constructor = Dog` Restores the correct constructor reference (it otherwise points to `Animal`).
4. **Creating an Instance** When `new Dog()` is called:
   - The instance gets its own `name` and `breed` properties.
   - The prototype chain allows it to access methods from both `Dog.prototype` and `Animal.prototype`.

3.4.4 Visualizing the Inheritance Chain

`myDog → Dog.prototype → Animal.prototype → Object.prototype → null`

- `myDog.bark()` → found on `Dog.prototype`
- `myDog.speak()` → not found on `Dog.prototype`, but found on `Animal.prototype`

3.4.5 Why Use Prototype Inheritance?

- **Reusability**: Common methods like `speak()` can be written once and shared across many types of animals.
- **Extendability**: Subtypes like `Dog`, `Cat`, etc., can have specialized methods while still reusing base functionality.
- **Performance**: Shared methods are stored once on the prototype, rather than duplicated per instance.

3.4.6 Summary

Prototype-based inheritance allows objects to inherit from other objects by linking their prototypes. Using constructor functions and `Object.create()`, you can build a classic inheritance hierarchy where child constructors reuse and extend the behavior of their parents. This system, though replaced in modern code by `class` syntax, remains fundamental to understanding how JavaScript handles inheritance under the hood.

3.5 Practical: Prototype-Based Inheritance Example

Now that you understand how prototype-based inheritance works in theory, let's build a practical example step by step. We'll create a small inheritance hierarchy with a base `Animal` constructor and two derived constructors: `Dog` and `Cat`. This example will demonstrate shared methods, inheritance setup, method overriding, and how instances interact with the prototype chain.

3.5.1 Step 1: Define the Base Constructor `Animal`

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

Animal.prototype.speak = function() {
  console.log(`${this.name} makes a generic animal sound.`);
};
```

- `Animal` is our base constructor function.
- It sets the `name` property and provides a shared method `speak()` on the prototype.

3.5.2 Step 2: Define the Dog Constructor and Inherit from Animal

```
function Dog(name, breed) {
  Animal.call(this, name); // Call the parent constructor
  this.breed = breed;
}

// Set up inheritance
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Add or override methods
Dog.prototype.speak = function() {
  console.log(`${this.name} the ${this.breed} barks.`);
};
```

- Dog calls Animal to inherit the name property.
- Dog.prototype is linked to Animal.prototype for method inheritance.
- The speak() method is overridden to provide dog-specific behavior.

3.5.3 Step 3: Define the Cat Constructor and Inherit from Animal

```
function Cat(name, color) {
  Animal.call(this, name); // Call the parent constructor
  this.color = color;
}

// Set up inheritance
Cat.prototype = Object.create(Animal.prototype);
Cat.prototype.constructor = Cat;

// Add a new method without overriding
Cat.prototype.purr = function() {
  console.log(`${this.name} the ${this.color} cat purrs softly.`);
};
```

- Cat also inherits from Animal and adds a new method purr() instead of overriding speak().

3.5.4 Step 4: Create Instances and Test Behavior

```
const rex = new Dog("Rex", "Labrador");
const luna = new Cat("Luna", "gray");

rex.speak();   // Output: Rex the Labrador barks.
```

```
luna.speak();  // Output: Luna makes a generic animal sound.
luna.purr();   // Output: Luna the gray cat purrs softly.
```

3.5.5 Step 5: Verify the Prototype Chain

```
console.log(rex instanceof Dog);     // true
console.log(rex instanceof Animal);  // true
console.log(luna instanceof Cat);    // true
console.log(luna instanceof Animal); // true
```

- These checks confirm that `rex` and `luna` are instances of both their own constructors and the base `Animal` constructor.

Full runnable code:

```
// Step 1: Define the Base Constructor `Animal`
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  console.log(`${this.name} makes a generic animal sound.`);
};

// Step 2: Define the `Dog` Constructor and Inherit from `Animal`
function Dog(name, breed) {
  Animal.call(this, name); // Call the parent constructor
  this.breed = breed;
}

// Set up inheritance
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Override speak method
Dog.prototype.speak = function() {
  console.log(`${this.name} the ${this.breed} barks.`);
};

// Step 3: Define the `Cat` Constructor and Inherit from `Animal`
function Cat(name, color) {
  Animal.call(this, name); // Call the parent constructor
  this.color = color;
}

// Set up inheritance
Cat.prototype = Object.create(Animal.prototype);
Cat.prototype.constructor = Cat;

// Add a new method
Cat.prototype.purr = function() {
```

```
  console.log(`${this.name} the ${this.color} cat purrs softly.`);
};

// Step 4: Create Instances and Test Behavior
const rex = new Dog("Rex", "Labrador");
const luna = new Cat("Luna", "gray");

rex.speak();   // Output: Rex the Labrador barks.
luna.speak();  // Output: Luna makes a generic animal sound.
luna.purr();   // Output: Luna the gray cat purrs softly.

// Step 5: Verify the Prototype Chain
console.log(rex instanceof Dog);     // true
console.log(rex instanceof Animal);  // true
console.log(luna instanceof Cat);    // true
console.log(luna instanceof Animal); // true
```

3.5.6 Summary

This example demonstrates how to:

- Use `Animal.call(this, ...)` to inherit constructor logic.
- Use `Object.create()` to inherit from a prototype.
- Override inherited methods (like `Dog.prototype.speak`).
- Add unique methods to subclasses (like `Cat.prototype.purr`).
- Leverage the prototype chain to reuse and organize shared behavior.

Prototype-based inheritance is a core part of JavaScript's OOP capabilities. This pattern forms the foundation for more advanced patterns, including ES6 `class` syntax, which builds directly on this system.

Chapter 4.

Constructors and the `new` Keyword

1. Constructor Functions Explained
2. Using `new` to Create Object Instances
3. Properties and Methods in Constructors
4. Practical: Creating Multiple Instances of a Car Object

4 Constructors and the `new` Keyword

4.1 Constructor Functions Explained

In JavaScript, **constructor functions** are special functions used to create and initialize new objects. Before the introduction of ES6 `class` syntax, constructor functions were the primary way to define reusable object templates.

4.1.1 What Is a Constructor Function?

A constructor function is a **regular function** designed to be used with the `new` keyword. When called with `new`, it performs three key actions:

1. A new empty object is created.
2. The new object is linked to the constructor's prototype.
3. The function body is executed with `this` bound to the new object.
4. The new object is returned automatically (unless another object is explicitly returned).

4.1.2 Example: Basic Constructor Function

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

const alice = new Person("Alice", 30);
console.log(alice.name); // "Alice"
console.log(alice.age);  // 30
```

Here's what happens under the hood when `new Person("Alice", 30)` is called:

- A new object is created.
- `this.name = name` sets `name` on the new object.
- The object is returned and stored in `alice`.

4.1.3 Blueprint for Objects

Constructor functions act like **blueprints** for creating multiple objects with the same structure:

```
const bob = new Person("Bob", 25);
console.log(bob.name); // "Bob"
```

Each call to `new Person(...)` creates a new `Person` object with its own `name` and `age`.

4.1.4 Naming Convention

By convention, constructor function names are **capitalized** (e.g., `Person`, `Car`, `Animal`) to distinguish them from regular functions. This signals to other developers that the function is intended to be used with `new`.

```
function car(make, model) { // NO Not recommended
  this.make = make;
  this.model = model;
}

function Car(make, model) { // YES Preferred
  this.make = make;
  this.model = model;
}
```

4.1.5 How Constructor Functions Differ from Regular Functions

Feature	Constructor Function (`new`)	Regular Function (without `new`)
Called with `new`?	Yes	No
Creates a new object?	Yes	No
Sets `this` to object?	Yes	Usually global or `undefined`
Returns object?	Yes (automatically)	Only if explicitly returned

4.1.6 Common Pitfalls

Forgetting `new`

If you call a constructor function without `new`, `this` will not refer to a new object. It may refer to the global object (non-strict mode) or `undefined` (strict mode), leading to unexpected bugs.

```
const badPerson = Person("Eve", 40); // NO No 'new' used
console.log(badPerson); // undefined
```

To avoid this, always use `new` when calling a constructor.

Returning a Different Object

If a constructor function explicitly returns an object, that object is returned instead of the one created by `new`:

```
function WeirdPerson(name) {
  this.name = name;
  return { greeting: "Hi" }; // overrides the created object
}

const wp = new WeirdPerson("Sam");
console.log(wp.name); // undefined
console.log(wp.greeting); // "Hi"
```

Usually, you should avoid returning objects from constructors unless you intend to override the default return.

4.1.7 Summary

- Constructor functions are reusable blueprints for creating similar objects.
- They work in tandem with the `new` keyword to automate object creation.
- Capitalize constructor names to follow convention and improve readability.
- Constructors were the foundation of object creation in JavaScript before ES6 classes.

Understanding constructor functions is essential, as modern `class` syntax builds directly upon them. They also help clarify how JavaScript implements object creation and inheritance behind the scenes.

4.2 Using `new` to Create Object Instances

In JavaScript, the `new` keyword is essential when using **constructor functions** to create objects. It automates several behind-the-scenes steps that allow the function to behave like a class, producing consistent and properly-linked instances.

4.2.1 What Does `new` Do?

When you call a constructor function with `new`, JavaScript performs **four key steps**:

```
const obj = new ConstructorFunction();
```

Here's what happens under the hood:

1. **A new empty object is created**:
   ```
   const obj = {};
   ```
2. **The new object is linked to the constructor's prototype**:
   ```
   Object.setPrototypeOf(obj, ConstructorFunction.prototype);
   ```
3. **The constructor function is called with `this` bound to the new object**:
   ```
   ConstructorFunction.call(obj);
   ```
4. **The new object is returned (unless another object is returned explicitly)**:
   ```
   return obj;
   ```

4.2.2 Example: Creating an Object with `new`

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

const alice = new Person("Alice", 30);
console.log(alice.name); // "Alice"
console.log(alice.age);  // 30
```

- `Person` is called as a constructor.
- A new object (`alice`) is created and initialized with `name` and `age`.
- `alice.__proto__` is set to `Person.prototype`.

4.2.3 Visual Breakdown

```
function Animal(type) {
  this.type = type;
}

const dog = new Animal("Dog");
```

Behind the scenes:

1. `const dog = {};`

2. `dog.__proto__ = Animal.prototype;`
3. `Animal.call(dog, "Dog");`
4. `return dog;`

4.2.4 Confirming the Prototype Link

```
console.log(dog instanceof Animal);             // true
console.log(Object.getPrototypeOf(dog) === Animal.prototype); // true
```

This confirms that `dog` inherits from `Animal.prototype`.

4.2.5 What Happens if You Forget `new`?

If you call a constructor function without `new`, `this` will not refer to a new object:

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

const wrongUser = User("Eve"); // NO no 'new'

console.log(wrongUser);       // undefined
console.log(global.name);     // "Eve" (in non-strict mode)
```

- In non-strict mode, `this` refers to the global object (e.g., `window` in browsers).
- In strict mode, `this` is `undefined`, and assigning `this.name` throws an error.

YES **Best practice**: Always use `new` with constructor functions to avoid subtle bugs.

4.2.6 Defensive Programming Tip

To prevent misuse, constructor functions can include a guard:

```
function Product(name) {
  if (!(this instanceof Product)) {
    return new Product(name); // auto-correct if called without new
  }
  this.name = name;
}
```

4.2.7 Summary

- The `new` keyword automates object creation from constructor functions.
- It links the new object to the prototype, binds `this`, and returns the instance.
- Forgetting `new` can result in incorrect context and subtle bugs.
- Constructor functions rely on `new` to behave properly—understanding its steps is essential for mastering JavaScript OOP.

4.3 Properties and Methods in Constructors

Constructor functions in JavaScript allow you to define **properties and methods** for each object instance by assigning them directly to `this`. This approach is straightforward and ensures that every object has its own unique set of data and behavior.

4.3.1 Defining Properties with `this`

To create instance-specific properties, assign values to `this` inside the constructor function:

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

const alice = new Person("Alice", 30);
console.log(alice.name); // "Alice"
```

Each call to `new Person()` creates a **new object** with its own `name` and `age`.

4.3.2 Defining Methods Inside Constructors

You can also define methods by assigning a function to `this`:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.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."
```

4.3.3 Important: Method Duplication

While this works, defining methods inside constructors **duplicates the function** for each instance:

```
const person1 = new Person("Amy", 40);
const person2 = new Person("Tom", 22);

console.log(person1.greet === person2.greet); // false
```

- Each instance gets its own copy of `greet`.
- This uses more memory and is **less efficient**, especially when many objects are created.

4.3.4 Best Practice: Use the Prototype for Shared Methods

Instead of defining methods in the constructor, place them on the prototype so all instances can share one copy:

```
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 person1 = new Person("Amy", 40);
const person2 = new Person("Tom", 22);

console.log(person1.greet === person2.greet); // true
```

YES **More memory efficient** and better for performance.

4.3.5 When to Use In-Constructor Methods

While prototype methods are preferred for shared behavior, in-constructor methods may be useful when:

- You need a **unique method** per instance (e.g., closures or private data).
- You're creating only a few objects and performance isn't a concern.

4.3.6 Summary

- Use `this.property = value` to assign instance-specific properties in constructors.
- Defining methods with `this.method = function() {}` gives each instance a unique copy.
- For shared behavior, **prefer adding methods to the constructor's prototype**.
- Understanding this distinction is key to writing optimized, scalable object-oriented JavaScript.

4.4 Practical: Creating Multiple Instances of a Car Object

Now that you understand how constructor functions and the `new` keyword work, let's put it all together in a practical example. We'll create a `Car` constructor that lets us build multiple car objects with their own properties, while efficiently sharing common methods.

4.4.1 Step 1: Define the Constructor Function

We'll define a `Car` constructor with `make`, `model`, and `year` as properties:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Each new `Car` object will receive its own copy of these values.

4.4.2 Step 2: Add Shared Methods Using the Prototype

We want every car to be able to start and display its information, but we don't want to duplicate the methods for every instance. So, we define the methods on the prototype:

```
Car.prototype.start = function() {
  console.log(`${this.make} ${this.model} is starting...`);
};

Car.prototype.displayInfo = function() {
  console.log(`Make: ${this.make}, Model: ${this.model}, Year: ${this.year}`);
};
```

These methods are shared across all car instances.

4.4.3 Step 3: Create Multiple Car Instances

Let's create a few different cars:

```
const car1 = new Car("Toyota", "Corolla", 2020);
const car2 = new Car("Ford", "Mustang", 2022);
const car3 = new Car("Tesla", "Model 3", 2023);
```

Each object stores its own data, but they all use the same `start()` and `displayInfo()` methods from the prototype.

4.4.4 Step 4: Use the Methods

Now let's try out the methods:

```
car1.start();          // Toyota Corolla is starting...
car1.displayInfo();    // Make: Toyota, Model: Corolla, Year: 2020

car2.start();          // Ford Mustang is starting...
car2.displayInfo();    // Make: Ford, Model: Mustang, Year: 2022

car3.start();          // Tesla Model 3 is starting...
car3.displayInfo();    // Make: Tesla, Model: Model 3, Year: 2023
```

4.4.5 Step 5: Check Shared Methods

Let's verify that the methods are shared:

```
console.log(car1.start === car2.start); // true
```

This confirms that both instances use the **same function** for `start()`, which saves memory and improves performance.

Full runnable code:

```
// Step 1: Define the Constructor Function
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}

// Step 2: Add Shared Methods Using the Prototype
Car.prototype.start = function() {
  console.log(`${this.make} ${this.model} is starting...`);
};
```

```
Car.prototype.displayInfo = function() {
  console.log(`Make: ${this.make}, Model: ${this.model}, Year: ${this.year}`);
};

// Step 3: Create Multiple Car Instances
const car1 = new Car("Toyota", "Corolla", 2020);
const car2 = new Car("Ford", "Mustang", 2022);
const car3 = new Car("Tesla", "Model 3", 2023);

// Step 4: Use the Methods
car1.start();         // Toyota Corolla is starting...
car1.displayInfo();   // Make: Toyota, Model: Corolla, Year: 2020

car2.start();         // Ford Mustang is starting...
car2.displayInfo();   // Make: Ford, Model: Mustang, Year: 2022

car3.start();         // Tesla Model 3 is starting...
car3.displayInfo();   // Make: Tesla, Model: Model 3, Year: 2023

// Step 5: Check Shared Methods
console.log(car1.start === car2.start); // true
```

4.4.6 Summary

- The `Car` constructor creates new objects with their own `make`, `model`, and `year`.
- Common methods like `start()` and `displayInfo()` are added to the prototype, so they are shared among all instances.
- This approach balances **customized data per object** with **efficient, reusable behavior**, which is a key advantage of JavaScript's object-oriented design.

By combining constructors with prototype methods, you create scalable, maintainable object models—perfect for real-world applications.

Chapter 5.

ES6 Classes Syntax and Concepts

1. Introduction to ES6 Classes
2. Class Declarations vs. Constructor Functions
3. Instance Properties and Methods
4. Class Fields and Static Members
5. Practical: Defining a Bank Account Class

5 ES6 Classes Syntax and Concepts

5.1 Introduction to ES6 Classes

With the release of ES6 (ECMAScript 2015), JavaScript introduced the **class syntax**—a new, cleaner way to create objects and manage inheritance. Though often called "classes," they are primarily **syntactic sugar** over the existing prototype-based system, designed to make object-oriented programming more intuitive and readable.

5.1.1 What Are ES6 Classes?

Before ES6, JavaScript relied on constructor functions and prototypes to create objects and establish inheritance. This approach works well but can sometimes feel verbose or confusing for developers coming from class-based languages like Java or C++.

ES6 classes provide a **more structured and familiar syntax** that wraps these prototype-based mechanisms behind a clear, concise interface.

5.1.2 Basic Structure of a Class

An ES6 class defines a blueprint for creating objects using the `class` keyword:

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

  greet() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  }
}
```

- The `constructor` method is a special function called when creating new instances.
- Methods like `greet()` are added to the class prototype behind the scenes.
- The class body groups properties and methods neatly, improving readability.

5.1.3 How Classes Relate to Functions and Prototypes

Despite the new syntax, **classes are still functions under the hood**:

```
typeof Person; // "function"
```

When you define a class, JavaScript creates a constructor function and sets up the prototype chain automatically. Using `new Person()` works exactly like before, but the class syntax hides the complexity.

5.1.4 Key Benefits of ES6 Classes

- **Readability**: Clear, familiar syntax that mirrors other OOP languages.
- **Organization**: Group related properties and methods in one place.
- **Inheritance**: `extends` keyword enables simple subclassing.
- **Strict Mode**: Classes run in strict mode by default, reducing errors.

5.1.5 Summary

ES6 classes do not introduce a new object model but provide a more elegant and accessible way to create and manage objects and inheritance. They improve code clarity, making JavaScript OOP easier to learn and maintain, while still leveraging the power of prototypes behind the scenes.

5.2 Class Declarations vs. Constructor Functions

ES6 classes and traditional constructor functions both enable object creation and inheritance in JavaScript, but they differ in syntax, behavior, and some internal rules. Understanding these differences helps you choose the right approach and write clearer, more maintainable code.

5.2.1 Syntax Differences

Constructor Function:

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

Person.prototype.greet = function() {
```

```
  console.log(`Hi, I'm ${this.name}.`);
};
```

ES6 Class Declaration:

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

  greet() {
    console.log(`Hi, I'm ${this.name}.`);
  }
}
```

- Classes use the `class` keyword and group constructor and methods inside a class body.
- Methods inside a class are added to the prototype implicitly—no need for explicit prototype assignments.
- The `constructor` method inside a class replaces the constructor function.

5.2.2 Hoisting Behavior

- **Constructor functions** are hoisted like regular functions, meaning you can call them before their definition in the code:

```
const p = new Person("Alice", 30); // Works even if Person defined later

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

- **Class declarations** are **not hoisted**. Trying to instantiate a class before its declaration results in a `ReferenceError`:

```
const p = new Person("Bob", 25); // ReferenceError: Cannot access 'Person' before initialization

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

This behavior encourages more predictable and organized code.

5.2.3 Enforcing `new` Usage

- **Constructor functions** can be called **without** `new`, which can lead to bugs because `this` won't refer to a new object:

```
function Animal(type) {
  this.type = type;
}

const wrong = Animal("Cat"); // `this` is global or undefined, no new object created
```

- **Classes must** be called with `new`. Calling a class constructor without `new` throws a `TypeError` immediately:

```
class Animal {
  constructor(type) {
    this.type = type;
  }
}

const wrong = Animal("Cat"); // TypeError: Class constructor Animal cannot be invoked without 'ne
```

This built-in safeguard improves reliability.

5.2.4 Similarities

- Both create constructor functions under the hood.
- Both support prototype-based inheritance.
- Instances created by either method behave similarly with respect to the prototype chain.
- Methods defined in class bodies or on the prototype are shared by instances.

5.2.5 Summary

Feature	Constructor Functions	ES6 Classes
Syntax	Function + prototype assignments	`class` keyword with methods inside
Hoisting	Function declarations are hoisted	Classes are not hoisted
Enforce `new`	No, can be called without `new`	Yes, must be called with `new`
Prototype method syntax	Explicit `Constructor.prototype.method = ...`	Implicit via class method definitions
Behavior under the hood	Functions with prototypes	Functions with prototypes

ES6 classes provide cleaner, more robust syntax and enforce good practices, while constructor functions offer flexibility and backward compatibility. Understanding both deepens your mastery of JavaScript's object-oriented capabilities.

5.3 Instance Properties and Methods

In ES6 classes, **instance properties** and **methods** are defined in a clear and organized way using the `constructor()` method and the class body. Understanding how to set up these elements is key to creating robust and maintainable objects.

5.3.1 Defining Instance Properties in the `constructor()`

The `constructor()` method is a special method called automatically when a new instance of the class is created using `new`. It is the place where you typically **initialize instance properties**.

```
class Person {
  constructor(name, age) {
    this.name = name; // Instance property
    this.age = age;   // Instance property
  }
}
```

Here, `this.name` and `this.age` become properties unique to each `Person` instance.

5.3.2 Defining Instance Methods in the Class Body

Methods declared **inside the class body but outside the constructor** are automatically added to the class's **prototype**. This means all instances share these methods instead of each having their own copy, which is memory efficient.

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

  greet() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  }
}
```

The `greet()` method above is shared by all instances of `Person`.

5.3.3 Creating and Using Instances

```
const alice = new Person("Alice", 30);
alice.greet(); // Output: Hi, I'm Alice and I'm 30 years old.
```

Each instance has its own set of properties (`name` and `age`), but they all use the **same** `greet()` method defined on the prototype.

5.3.4 Summary of Behavior

Definition Location	Stored On	Shared Among Instances?
Properties defined in `constructor()` via `this`	On each individual instance	No, unique per object
Methods defined in class body (outside constructor)	On the class prototype	Yes, shared by all instances

5.3.5 Why This Matters

- Initializing properties inside the constructor ensures that each object has its own data.
- Declaring methods in the class body keeps a single shared copy, saving memory.
- This structure makes your classes both **efficient** and **easy to understand**.

By following this pattern, ES6 classes provide a clean and natural way to manage both instance-specific data and shared behavior in your JavaScript applications.

5.4 Class Fields and Static Members

ES6 classes brought a cleaner way to define objects, but subsequent updates to JavaScript have introduced **class fields** and enhanced support for **static members**, making class syntax even more powerful and expressive.

5.4.1 Class Fields: Public and Private

Traditionally, instance properties are defined inside the `constructor()` using `this`. However, **class fields** allow you to declare properties directly within the class body, making code cleaner and more declarative.

Public Fields

Public fields are accessible on each instance:

```
class Person {
  name = "Unknown"; // public field with default value

  constructor(age) {
    this.age = age;
  }
}

const p = new Person(25);
console.log(p.name); // "Unknown"
console.log(p.age);  // 25
```

Here, `name` is a public instance field declared outside the constructor.

Private Fields (ES2022)

Private fields start with a `#` and are **only accessible within the class** — not from outside or subclasses:

```
class Person {
  #ssn; // private field

  constructor(name, ssn) {
    this.name = name;
    this.#ssn = ssn;
  }

  getSSN() {
    return this.#ssn;
  }
}

const person = new Person("Alice", "123-45-6789");
console.log(person.name);      // "Alice"
console.log(person.getSSN());  // "123-45-6789"
console.log(person.#ssn);      // SyntaxError: Private field '#ssn' must be declared in an enclosing cl
```

Private fields help enforce **data privacy and encapsulation** within classes.

5.4.2 Static Properties and Methods

Static members belong to the class itself, not to instances. They are useful for:

- Utility functions related to the class concept but not specific to any object.
- Constants shared across all instances.
- Factory methods that create instances in special ways.

Defining Static Members

Use the `static` keyword before a property or method:

```
class Calculator {
  static pi = 3.14159; // static property

  static calculateCircumference(radius) {
    return 2 * Calculator.pi * radius;
  }
}

console.log(Calculator.pi);                    // 3.14159
console.log(Calculator.calculateCircumference(5)); // 31.4159

const calc = new Calculator();
console.log(calc.pi);  // undefined - static members aren't on instances
```

- You **access static members directly on the class**, not on instances.
- Static methods can call other static members using `this`.

5.4.3 Summary

Member Type	Declared With	Belongs To	Access Example
Public Instance Field	`fieldName = value` inside class body	Each instance	`instance.fieldName`
Private Instance Field	`#fieldName = value` inside class body	Each instance (private)	`instance.#fieldName` (only inside class)
Static Property/Method	`static propertyName/methodName`	The class itself	`ClassName.propertyName` or `ClassName.methodName()`

5.4.4 Practical Takeaway

- Use **class fields** to declare instance properties upfront for cleaner code.

- Use **private fields** to encapsulate sensitive data.
- Use **static members** for utility methods and constants that belong to the class, not to individual instances.

These features together provide more expressive, safe, and maintainable object-oriented code in modern JavaScript.

5.5 Practical: Defining a Bank Account Class

Let's put together the concepts from this chapter by creating a `BankAccount` class. This class will demonstrate how to use instance properties, methods, class fields, and static members in a real-world scenario.

5.5.1 Step 1: Define the Class with Class Fields

We want each `BankAccount` instance to have:

- An `accountNumber` (unique for each account)
- A `balance` (starting at zero)

We'll also include methods to deposit and withdraw money and to check the balance.

```
class BankAccount {
  // Static field to keep track of the last assigned account number
  static lastAccountNumber = 1000;

  // Instance fields with default values
  accountNumber;
  balance = 0;

  constructor(initialBalance = 0) {
    this.accountNumber = BankAccount.generateAccountNumber();
    this.balance = initialBalance;
  }

  // Static method to generate a unique account number
  static generateAccountNumber() {
    return ++BankAccount.lastAccountNumber;
  }

  // Deposit money into the account
  deposit(amount) {
    if (amount <= 0) {
      console.log("Deposit amount must be positive.");
      return;
    }
    this.balance += amount;
```

```
    console.log(`Deposited $${amount}. New balance: $${this.balance}.`);
  }

  // Withdraw money from the account
  withdraw(amount) {
    if (amount <= 0) {
      console.log("Withdrawal amount must be positive.");
      return;
    }
    if (amount > this.balance) {
      console.log("Insufficient funds.");
      return;
    }
    this.balance -= amount;
    console.log(`Withdrew $${amount}. New balance: $${this.balance}.`);
  }

  // Get current balance
  getBalance() {
    return this.balance;
  }
}
```

5.5.2 Step 2: Create Instances and Use Methods

```
const account1 = new BankAccount(500);
console.log(`Account #${account1.accountNumber} created with balance $${account1.getBalance()}.`);

account1.deposit(150);  // Deposited $150. New balance: $650.
account1.withdraw(200); // Withdrew $200. New balance: $450.
account1.withdraw(500); // Insufficient funds.

const account2 = new BankAccount();
console.log(`Account #${account2.accountNumber} created with balance $${account2.getBalance()}.`);

account2.deposit(1000); // Deposited $1000. New balance: $1000.
```

Full runnable code:

```
// Step 1: Define the Class with Class Fields
class BankAccount {
  // Static field to keep track of the last assigned account number
  static lastAccountNumber = 1000;

  // Instance fields with default values
  accountNumber;
  balance = 0;

  constructor(initialBalance = 0) {
    this.accountNumber = BankAccount.generateAccountNumber();
    this.balance = initialBalance;
```

```
  }

  // Static method to generate a unique account number
  static generateAccountNumber() {
    return ++BankAccount.lastAccountNumber;
  }

  // Deposit money into the account
  deposit(amount) {
    if (amount <= 0) {
      console.log("Deposit amount must be positive.");
      return;
    }
    this.balance += amount;
    console.log(`Deposited $${amount}. New balance: $${this.balance}.`);
  }

  // Withdraw money from the account
  withdraw(amount) {
    if (amount <= 0) {
      console.log("Withdrawal amount must be positive.");
      return;
    }
    if (amount > this.balance) {
      console.log("Insufficient funds.");
      return;
    }
    this.balance -= amount;
    console.log(`Withdrew $${amount}. New balance: $${this.balance}.`);
  }

  // Get current balance
  getBalance() {
    return this.balance;
  }
}

// Step 2: Create Instances and Use Methods
const account1 = new BankAccount(500);
console.log(`Account #${account1.accountNumber} created with balance $${account1.getBalance()}.`);

account1.deposit(150);  // Deposited $150. New balance: $650.
account1.withdraw(200); // Withdrew $200. New balance: $450.
account1.withdraw(500); // Insufficient funds.

const account2 = new BankAccount();
console.log(`Account #${account2.accountNumber} created with balance $${account2.getBalance()}.`);

account2.deposit(1000); // Deposited $1000. New balance: $1000.
```

5.5.3 Explanation

- **Static field `lastAccountNumber`** tracks the last assigned account number shared across all instances.
- **Static method `generateAccountNumber()`** increments and returns a new unique number.
- **Instance fields** like `accountNumber` and `balance` store data unique to each account.
- Methods `deposit()`, `withdraw()`, and `getBalance()` provide safe ways to interact with the account balance.
- Validation inside methods prevents invalid operations, like withdrawing more than the current balance.

5.5.4 Summary

This example highlights:

- How to use **class fields** for instance properties.
- How **static fields and methods** support shared class-level data and utilities.
- Organizing instance behavior inside class methods.
- Providing a real-world example of OOP concepts in modern JavaScript.

This pattern forms a strong foundation for more complex object-oriented applications.

Chapter 6.

Inheritance with ES6 Classes

1. Extending Classes
2. Using `super` in Derived Classes
3. Overriding Methods
4. Practical: Modeling Animals with a Base and Derived Classes

6 Inheritance with ES6 Classes

6.1 Extending Classes

One of the key features of object-oriented programming is **inheritance**—the ability for one class to derive properties and methods from another. ES6 classes provide a straightforward way to achieve this using the `extends` keyword.

6.1.1 What Does `extends` Do?

When a class **extends** another class, it creates a **subclass** (or child class) that inherits all the properties and methods of the **parent class** (also called the base class). This means the subclass can reuse existing functionality and add or customize its own features.

6.1.2 Basic Syntax of Class Extension

Here's a simple example showing how to extend a class:

```
// Parent class
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

// Subclass extending Animal
class Dog extends Animal {
  bark() {
    console.log(`${this.name} barks.`);
  }
}

const myDog = new Dog("Buddy");
myDog.speak(); // Buddy makes a sound.
myDog.bark();  // Buddy barks.
```

- `Dog` inherits the `name` property and `speak()` method from `Animal`.
- `Dog` adds a new method `bark()`.
- An instance of `Dog` can access both inherited and its own methods.

6.1.3 How Inheritance Works Under the Hood

- The subclass's prototype internally links to the parent class's prototype.
- When a method or property is called on the subclass instance, JavaScript first looks in the subclass. If it's not found, it continues up the prototype chain to the parent class.

This prototype chain ensures seamless reuse and extension of functionality.

6.1.4 Adding New Functionality in Subclasses

Subclasses can not only inherit but also extend the behavior of the parent by adding new methods or properties:

```
// Parent class
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}
class Cat extends Animal {
  purr() {
    console.log(`${this.name} purrs.`);
  }
}

const kitty = new Cat("Whiskers");
kitty.speak(); // Whiskers makes a sound.
kitty.purr();  // Whiskers purrs.
```

Here, `Cat` adds a `purr()` method without affecting the parent `Animal`.

6.1.5 Summary

- Use `extends` to create subclasses that inherit from parent classes.
- Subclasses automatically gain access to the parent's properties and methods.
- You can add new properties and methods to customize subclass behavior.
- This feature promotes code reuse and logical organization of related classes.

Extending classes with `extends` makes JavaScript OOP more powerful and intuitive, enabling developers to build rich inheritance hierarchies cleanly and efficiently.

6.2 Using `super` in Derived Classes

In ES6 classes, the `super` keyword plays a crucial role when working with inheritance. It allows a subclass (derived class) to access and invoke methods or constructors from its parent class, ensuring proper initialization and method reuse.

6.2.1 Calling the Parent Constructor with `super()`

When you extend a class and define a constructor in the subclass, you **must** call `super()` before you use `this`. This is because the parent class's constructor needs to run first to set up the inherited properties properly.

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

class Dog extends Animal {
  constructor(name, breed) {
    super(name);   // Call parent constructor to initialize `name`
    this.breed = breed; // Initialize subclass-specific property
  }
}
```

- The `super(name)` call invokes the `Animal` constructor with the argument `name`.
- You cannot access `this.breed` or any other `this` property before calling `super()` — doing so throws a ReferenceError.

6.2.2 Using `super` to Call Parent Methods

Besides constructors, you can also use `super` to invoke methods defined in the parent class. This is especially useful when **overriding** a method in the subclass but still wanting to leverage the parent's implementation.

```
class Animal {
  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  speak() {
    super.speak();  // Call the parent class's speak()
    console.log(`${this.name} barks.`);
  }
```

```
}

const dog = new Dog("Buddy");
dog.speak();
// Output:
// Buddy makes a sound.
// Buddy barks.
```

- Here, `super.speak()` calls the `speak` method from `Animal`.
- The subclass then adds its own behavior after calling the parent method.

6.2.3 Summary of Key Points

Aspect	Description
`super()` in constructor	Must be called before accessing `this` in subclass constructor to initialize parent properties.
`super.methodName()`	Calls the parent class's method from the subclass, often used when overriding methods.
Order of execution	Parent constructor runs first, then subclass code executes.

6.2.4 Why Use `super`?

- Ensures the parent class's setup runs correctly.
- Enables code reuse and extension by building on existing parent class methods.
- Prevents duplication by allowing subclasses to augment rather than replace functionality.

Using `super` effectively ensures your subclasses integrate seamlessly with their parent classes, promoting clean and maintainable inheritance hierarchies in modern JavaScript applications.

6.3 Overriding Methods

One of the most powerful features of inheritance is **method overriding** — the ability of a subclass to provide its own version of a method that exists in the parent class. Overriding lets subclasses change or extend inherited behavior to suit their specific needs.

6.3.1 What Is Method Overriding?

When a subclass defines a method with the same name as a method in its parent class, the subclass's method **replaces** the parent's method for instances of the subclass. This means the subclass controls what happens when that method is called.

6.3.2 Basic Example of Overriding

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

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

const genericAnimal = new Animal();
genericAnimal.speak(); // Animal makes a sound.

const myDog = new Dog();
myDog.speak();         // Dog barks.
```

Here, the `Dog` class overrides the `speak()` method to provide behavior specific to dogs.

6.3.3 Calling the Parent Method with `super.methodName()`

Sometimes, you want to **extend** the behavior of the parent method rather than completely replace it. You can do this by calling the parent method inside the overridden method using `super`.

```
class Animal {
  speak() {
    console.log("Animal makes a sound.");
  }
}
class Dog extends Animal {
  speak() {
    super.speak();         // Call the parent class method
    console.log("Dog barks."); // Add extra behavior
  }
}
```

```
const myDog = new Dog();
myDog.speak();
// Output:
// Animal makes a sound.
// Dog barks.
```

This approach promotes **code reuse** and keeps the behavior consistent.

6.3.4 Best Practices for Overriding

- **Maintain Polymorphism**: Ensure overridden methods have compatible signatures and expected behavior so that instances of subclasses can be used interchangeably with parent class instances.
- **Use `super` Wisely**: Call the parent method if you want to build upon its logic, but omit it if you intend to completely replace the behavior.
- **Keep Code Clear**: Document overridden methods to clarify differences and extensions to inherited behavior.

6.3.5 Summary

- Overriding allows subclasses to tailor inherited methods to their needs.
- Use `super.methodName()` within overridden methods to incorporate parent class functionality.
- Proper overriding supports **polymorphism**, making your OOP designs flexible and maintainable.

Mastering method overriding is essential for building rich, dynamic class hierarchies in JavaScript's ES6+ object-oriented programming.

6.4 Practical: Modeling Animals with a Base and Derived Classes

Let's bring together what we've learned about ES6 class inheritance by modeling a simple animal hierarchy. We'll create a base `Animal` class and then extend it with `Dog` and `Cat` subclasses, demonstrating inheritance, method overriding, and adding specialized behaviors.

6.4.1 Step 1: Define the Base `Animal` Class

Our `Animal` class will have a `name` property and a generic `speak()` method:

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

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}
```

6.4.2 Step 2: Create the `Dog` Class Extending `Animal`

The `Dog` class will inherit from `Animal`, override `speak()`, and add a new method `fetch()`:

```
class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }

  fetch() {
    console.log(`${this.name} is fetching the ball!`);
  }
}
```

6.4.3 Step 3: Create the `Cat` Class Extending `Animal`

The `Cat` class also inherits from `Animal`, overrides `speak()`, and adds a new method `purr()`:

```
class Cat extends Animal {
  speak() {
    console.log(`${this.name} meows.`);
  }

  purr() {
    console.log(`${this.name} is purring.`);
  }
}
```

6.4.4 Step 4: Create Instances and Test Behavior

```
const genericAnimal = new Animal("Generic");
genericAnimal.speak();  // Output: Generic makes a sound.

const dog = new Dog("Buddy");
dog.speak();            // Output: Buddy barks.
dog.fetch();            // Output: Buddy is fetching the ball!

const cat = new Cat("Whiskers");
cat.speak();            // Output: Whiskers meows.
cat.purr();             // Output: Whiskers is purring.
```

Full runnable code:

```
// Step 1: Define the Base `Animal` Class
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

// Step 2: Create the `Dog` Class Extending `Animal`
class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }

  fetch() {
    console.log(`${this.name} is fetching the ball!`);
  }
}

// Step 3: Create the `Cat` Class Extending `Animal`
class Cat extends Animal {
  speak() {
    console.log(`${this.name} meows.`);
  }

  purr() {
    console.log(`${this.name} is purring.`);
  }
}

// Step 4: Create Instances and Test Behavior
const genericAnimal = new Animal("Generic");
genericAnimal.speak();  // Output: Generic makes a sound.

const dog = new Dog("Buddy");
dog.speak();            // Output: Buddy barks.
dog.fetch();            // Output: Buddy is fetching the ball!
```

```
const cat = new Cat("Whiskers");
cat.speak();            // Output: Whiskers meows.
cat.purr();             // Output: Whiskers is purring.
```

6.4.5 Whats Happening Here?

- **Inheritance**: Both `Dog` and `Cat` classes inherit the `name` property from `Animal`.
- **Method Overriding**: `Dog` and `Cat` override the `speak()` method to provide species-specific sounds.
- **Specialized Behavior**: Each subclass adds its own unique method (`fetch` for `Dog` and `purr` for `Cat`).
- **Instance Creation**: Instances of each class maintain their own state (`name`) and have access to appropriate behaviors.

Chapter 7.

Encapsulation and Data Privacy

1. Private Properties and Methods (ES2022+ Private Fields)
2. Using Closures for Encapsulation
3. Getters and Setters
4. Practical: Creating a Secure User Account Class

7 Encapsulation and Data Privacy

7.1 Private Properties and Methods (ES2022+ Private Fields)

Encapsulation is a fundamental principle of object-oriented programming (OOP). It means **hiding the internal details** of an object to protect its internal state and control how it can be accessed or modified. By restricting access to certain properties or methods, encapsulation helps prevent unintended interference and makes code more secure and maintainable.

7.1.1 The Challenge of Encapsulation in JavaScript

Traditionally, JavaScript did not provide built-in syntax for private properties or methods. Developers used naming conventions (like prefixing with an underscore `_property`) or closures to simulate privacy. However, these approaches were not enforced by the language, and properties remained publicly accessible if someone knew the name.

7.1.2 Introducing Private Fields and Methods with

Since ES2022, JavaScript supports **true private fields and methods** using the `#` syntax. These private members are only accessible within the class body, and attempts to access them from outside the class will result in a syntax error.

This means privacy is now enforced by the language itself — a much stronger guarantee than conventions or closures.

7.1.3 How to Declare Private Fields and Methods

- Private fields and methods are declared by prefixing their name with a `#`.
- They can only be accessed inside the class using the same `#` prefix.
- They are completely hidden from instances and outside code.

```
class User {
  #password; // Private field

  constructor(name, password) {
    this.name = name;     // Public property
    this.#password = password; // Private property
  }

  // Private method
```

```
  #validatePassword(pw) {
    return pw.length >= 8;
  }

  // Public method accessing private members
  changePassword(newPassword) {
    if (this.#validatePassword(newPassword)) {
      this.#password = newPassword;
      console.log("Password changed successfully.");
    } else {
      console.log("Password too short!");
    }
  }
}

const user = new User("Alice", "secret123");
console.log(user.name);       // Alice
//console.log(user.#password);  // SyntaxError: Private field '#password' must be declared in an enclos
user.changePassword("123");   // Password too short!
```

7.1.4 Key Differences Between Private and Public Members

Aspect	Public Members	Private Members (using `#`)
Accessibility	Accessible anywhere	Accessible **only** inside the class
Syntax	`this.property`	`this.#property`
Security	No enforced privacy	Privacy enforced by the language
Reflection	Shows up in `Object.keys()` and enumeration	Hidden, not enumerable

7.1.5 Benefits of Using Private Fields and Methods

- **Stronger Encapsulation**: You can confidently hide sensitive data and implementation details.
- **Improved Security**: Private data cannot be tampered with or accessed unintentionally.
- **Cleaner APIs**: Only expose what's necessary, keeping the class interface simple.
- **Better Maintainability**: Internal changes won't affect outside code, reducing bugs.

7.1.6 Summary

The `#` syntax for private fields and methods is a modern JavaScript feature that brings **true encapsulation** to your classes. By using private members, you protect your object's internal

state and ensure your code is more robust, secure, and easier to maintain. This approach represents a significant improvement over older patterns and is a best practice for managing data privacy in contemporary JavaScript OOP.

7.2 Using Closures for Encapsulation

Before JavaScript introduced **private fields** with the # syntax, developers often relied on **closures** to simulate private properties and methods. Closures allow functions to **remember** and access variables from their surrounding scope even after that outer function has finished executing, providing a powerful way to encapsulate data.

7.2.1 What Are Closures?

A **closure** is created when a function "closes over" variables from its **lexical environment** (the surrounding scope where it was defined). This means inner functions can access variables declared in their outer function, even if the outer function has returned.

This feature allows us to keep data private by storing it in the outer function's scope and exposing only specific methods that can access or modify it.

7.2.2 Using Closures to Simulate Private Properties

By defining variables inside a constructor function (or factory function) but **not attaching them to `this`**, those variables remain hidden from the outside. Only privileged methods defined within the same scope can access or modify these variables.

```
function User(name, password) {
  // Private variables inside the closure
  let _password = password;

  this.name = name; // Public property

  // Privileged method with access to _password
  this.checkPassword = function(input) {
    return input === _password;
  };

  this.changePassword = function(newPassword) {
    if (newPassword.length >= 8) {
      _password = newPassword;
      console.log("Password changed.");
    } else {
```

```
      console.log("Password too short!");
    }
  };
}

const user = new User("Alice", "secret123");
console.log(user.name);             // Alice
console.log(user._password);        // undefined (not accessible)
console.log(user.checkPassword("secret123"));  // true
user.changePassword("123");         // Password too short!
user.changePassword("newpassword"); // Password changed.
```

7.2.3 How This Works

- `_password` is a **local variable** inside the `User` function, **not a property** of the returned object.
- Because `_password` is not exposed via `this`, code outside the constructor cannot access it directly.
- The methods `checkPassword` and `changePassword` are **privileged methods** — they have access to `_password` because of the closure.
- Even after the `User` constructor finishes, the privileged methods retain access to `_password`.

7.2.4 Advantages and Limitations

Advantages	Limitations
True privacy: outside code cannot access private data directly	Methods are recreated for every instance (memory cost)
Flexibility in controlling access to data	Cannot use inheritance easily with this pattern
No reliance on naming conventions	Slightly more complex syntax

7.2.5 Summary

Using closures to encapsulate data is a powerful technique in JavaScript that predates native private fields. By defining private variables inside a function and exposing only specific methods to interact with them, you can effectively **hide internal state** and protect your objects from external modification. This pattern is foundational for understanding JavaScript's approach to privacy and remains useful in many scenarios today.

7.3 Getters and Setters

In object-oriented programming, **getters** and **setters** are special methods that allow controlled access to an object's properties while preserving encapsulation. Instead of directly accessing or modifying a property, getters and setters provide a way to **intercept** these operations, enabling validation, computed values, logging, or other side effects.

7.3.1 Why Use Getters and Setters?

- **Validation**: Ensure only valid data is assigned to a property.
- **Computed Properties**: Return values derived from other properties.
- **Encapsulation**: Hide the internal representation and expose a clean interface.
- **Logging/Debugging**: Track when properties are accessed or changed.

7.3.2 Getters and Setters in ES6 Classes

In ES6 classes, you define getters and setters using the `get` and `set` keywords inside the class body:

```
class Person {
  constructor(firstName, lastName) {
    this._firstName = firstName;  // Conventionally private property
    this._lastName = lastName;
  }

  // Getter for fullName (computed property)
  get fullName() {
    return `${this._firstName} ${this._lastName}`;
  }

  // Setter for fullName with validation
  set fullName(name) {
    const parts = name.split(" ");
    if (parts.length === 2) {
      this._firstName = parts[0];
      this._lastName = parts[1];
    } else {
      console.log("Invalid full name format.");
    }
  }
}

const person = new Person("John", "Doe");
console.log(person.fullName);  // John Doe

person.fullName = "Jane Smith";
console.log(person.fullName);  // Jane Smith
```

```
person.fullName = "InvalidName";  // Invalid full name format.
```

- The getter `fullName` lets you access a **computed** property.
- The setter `fullName` allows you to **validate** and split the full name into parts.
- Note that internal properties are named with an underscore _ by convention to indicate they should not be accessed directly.

7.3.3 Getters and Setters in Object Literals

You can also define getters and setters inside object literals using the same `get` and `set` keywords:

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

  get area() {
    return this.width * this.height;
  },

  set area(value) {
    console.log(`Setting area to ${value} is not allowed.`);
  }
};

console.log(rectangle.area);  // 50
rectangle.area = 100;          // Setting area to 100 is not allowed.
```

7.3.4 Summary

- **Getters** allow properties to be accessed like regular properties but can compute or control the returned value.
- **Setters** allow validation or other logic when properties are assigned.
- Both help maintain **encapsulation** by managing how internal data is accessed and modified.
- Use getters and setters to make your objects safer, more flexible, and easier to maintain.

By mastering getters and setters, you can create intuitive and powerful interfaces for your JavaScript objects, enhancing both usability and data integrity.

7.4 Practical: Creating a Secure User Account Class

In this section, we'll build a `UserAccount` class that demonstrates how to protect sensitive data—like passwords—using **private fields** and **encapsulation** techniques. We'll include methods that safely update and retrieve user information using **getters and setters**, ensuring controlled access and reinforcing best security practices.

7.4.1 Designing the `UserAccount` Class

Our goals are:

- **Hide sensitive data** such as the password using private fields.
- Use **getters** to retrieve user info safely without exposing sensitive data.
- Use **setters** to update information with validation.
- Provide methods to **change the password** securely.
- Demonstrate how encapsulation keeps data private and methods control interactions.

7.4.2 Implementation Using Private Fields (#)

```
class UserAccount {
  // Private fields
  #password;

  constructor(username, password, email) {
    this.username = username; // Public property
    this.email = email;       // Public property
    this.#password = password; // Private property
  }

  // Getter for username
  getUsername() {
    return this.username;
  }

  // Getter for email
  getEmail() {
    return this.email;
  }

  // Setter for email with simple validation
  setEmail(newEmail) {
    if (newEmail.includes("@")) {
      this.email = newEmail;
      console.log("Email updated.");
    } else {
      console.log("Invalid email format.");
```

```
    }
  }

  // Method to verify password (do not expose the password directly)
  verifyPassword(inputPassword) {
    return inputPassword === this.#password;
  }

  // Method to change password securely
  changePassword(oldPassword, newPassword) {
    if (!this.verifyPassword(oldPassword)) {
      console.log("Old password is incorrect.");
      return;
    }
    if (newPassword.length < 8) {
      console.log("New password must be at least 8 characters long.");
      return;
    }
    this.#password = newPassword;
    console.log("Password changed successfully.");
  }
}
```

7.4.3 Using the `UserAccount` Class

```
const user = new UserAccount("alice", "secret123", "alice@example.com");

console.log(user.getUsername());  // alice
console.log(user.getEmail());     // alice@example.com

user.setEmail("alice@newdomain.com");  // Email updated.
user.setEmail("invalidEmail");          // Invalid email format.

// Password verification
console.log(user.verifyPassword("wrongpass"));  // false
console.log(user.verifyPassword("secret123"));  // true

// Attempt to change password with wrong old password
user.changePassword("wrongpass", "newpassword123");  // Old password is incorrect.

// Attempt to change password with short new password
user.changePassword("secret123", "short");           // New password must be at least 8 characters long

// Successful password change
user.changePassword("secret123", "newpassword123");  // Password changed successfully.

// Trying to access password directly throws an error
//console.log(user.#password);  // SyntaxError: Private field '#password' must be declared in an enclos
```

Full runnable code:

```
class UserAccount {
  // Private fields
  #password;

  constructor(username, password, email) {
    this.username = username; // Public property
    this.email = email;       // Public property
    this.#password = password; // Private property
  }

  // Getter for username
  getUsername() {
    return this.username;
  }

  // Getter for email
  getEmail() {
    return this.email;
  }

  // Setter for email with simple validation
  setEmail(newEmail) {
    if (newEmail.includes("@")) {
      this.email = newEmail;
      console.log("Email updated.");
    } else {
      console.log("Invalid email format.");
    }
  }

  // Method to verify password (do not expose the password directly)
  verifyPassword(inputPassword) {
    return inputPassword === this.#password;
  }

  // Method to change password securely
  changePassword(oldPassword, newPassword) {
    if (!this.verifyPassword(oldPassword)) {
      console.log("Old password is incorrect.");
      return;
    }
    if (newPassword.length < 8) {
      console.log("New password must be at least 8 characters long.");
      return;
    }
    this.#password = newPassword;
    console.log("Password changed successfully.");
  }
}
//Using the `UserAccount` Class

const user = new UserAccount("alice", "secret123", "alice@example.com");

console.log(user.getUsername());  // alice
console.log(user.getEmail());     // alice@example.com

user.setEmail("alice@newdomain.com");  // Email updated.
user.setEmail("invalidEmail");          // Invalid email format.
```

```
// Password verification
console.log(user.verifyPassword("wrongpass"));  // false
console.log(user.verifyPassword("secret123"));  // true

// Attempt to change password with wrong old password
user.changePassword("wrongpass", "newpassword123");  // Old password is incorrect.

// Attempt to change password with short new password
user.changePassword("secret123", "short");           // New password must be at least 8 characters long

// Successful password change
user.changePassword("secret123", "newpassword123");  // Password changed successfully.

// Trying to access password directly throws an error
//console.log(user.#password);  // SyntaxError: Private field '#password' must be declared in an enclos
```

7.4.4 Explanation

- The `#password` field is **private**, so it cannot be accessed or modified directly from outside the class.
- Public methods like `verifyPassword` and `changePassword` control how the password is checked and updated.
- The `setEmail` method includes **validation** to ensure only valid emails are set.
- This approach keeps sensitive data **secure** and the object's interface **clean and safe**.

7.4.5 Summary

Using private fields and controlled access methods in the `UserAccount` class shows how JavaScript's modern features help implement **robust encapsulation**. This pattern protects sensitive data, enforces validation, and promotes best security practices in real-world applications. By applying these principles, your object-oriented JavaScript code will be both safer and easier to maintain.

Chapter 8.

Polymorphism in JavaScript

1. Understanding Polymorphism
2. Method Overriding and Dynamic Dispatch
3. Practical: Shape Drawing Example with Polymorphism

8 Polymorphism in JavaScript

8.1 Understanding Polymorphism

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different types to be treated through a common interface. The term literally means "many forms," reflecting how different objects can respond to the same method call in their own unique way.

8.1.1 What is Polymorphism?

Imagine you have different animals — a dog, a cat, and a bird. Each animal can **make a sound**, but the sound they make is different:

- A dog **barks**.
- A cat **meows**.
- A bird **chirps**.

Despite these differences, you can call the same method, like `makeSound()`, on any of these animals without worrying about their specific types. This is polymorphism in action: **one interface, multiple behaviors**.

8.1.2 Why Is Polymorphism Important?

Polymorphism allows for:

- **Flexibility:** Code that works with a general type or interface can handle new specific types without changes.
- **Extensibility:** New object types can be introduced without rewriting existing code, as long as they follow the expected interface.
- **Simpler Code:** Developers can write generic code that works with different objects, improving maintainability and reducing duplication.

For example, a drawing application might call `draw()` on various shape objects — circles, rectangles, triangles — and each shape knows how to draw itself correctly.

8.1.3 How Does JavaScript Achieve Polymorphism?

JavaScript is a **dynamically typed language**, meaning variables don't have fixed types. While it doesn't have classical interfaces like some statically typed languages, polymorphism

is naturally achieved through **shared method names** and **duck typing** — "If it looks like a duck and quacks like a duck, it's a duck."

As long as different objects implement methods with the same name, you can treat them interchangeably:

```
class Dog {
  makeSound() {
    console.log("Bark!");
  }
}

class Cat {
  makeSound() {
    console.log("Meow!");
  }
}

function animalSound(animal) {
  animal.makeSound();  // Works with any object that has makeSound()
}

const dog = new Dog();
const cat = new Cat();

animalSound(dog);  // Bark!
animalSound(cat);  // Meow!
```

8.1.4 Summary

Polymorphism enables objects of different types to be handled through a unified interface, promoting code reuse and extensibility. In JavaScript, this flexibility arises from its dynamic nature and the ability of different objects to implement methods with the same names. Understanding polymorphism empowers you to design systems where diverse objects can work together seamlessly.

8.2 Method Overriding and Dynamic Dispatch

In object-oriented programming, **method overriding** and **dynamic dispatch** are key mechanisms that enable polymorphism — allowing derived classes to customize or extend the behavior of methods defined in their parent classes.

8.2.1 What is Method Overriding?

Method overriding happens when a subclass (derived class) provides its own implementation of a method that is already defined in its parent class. This lets the subclass tailor or completely replace the behavior inherited from the parent.

For example, suppose you have a general `Animal` class with a method `speak()`. Different animals might override this method to produce their unique sounds:

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

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

class Cat extends Animal {
  speak() {
    console.log("Meow!");
  }
}
```

Here, `Dog` and `Cat` override the `speak()` method to provide specific implementations.

8.2.2 What is Dynamic Dispatch?

Dynamic dispatch is the runtime process that determines **which version of an overridden method to invoke** when a method call is made on an object. It means the program decides at execution time which method implementation to run based on the **actual type** of the object, not the type of the reference.

Consider this code:

```
function makeAnimalSpeak(animal) {
  animal.speak();  // Which speak() runs depends on the object passed in
}

const dog = new Dog();
const cat = new Cat();

makeAnimalSpeak(dog);  // Bark!
makeAnimalSpeak(cat);  // Meow!
```

Even though the parameter `animal` is generic, the call to `speak()` invokes the appropriate method defined in the specific subclass (`Dog` or `Cat`) due to dynamic dispatch.

Full runnable code:

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

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

class Cat extends Animal {
  speak() {
    console.log("Meow!");
  }
}
function makeAnimalSpeak(animal) {
  animal.speak();  // Which speak() runs depends on the object passed in
}

const dog = new Dog();
const cat = new Cat();

makeAnimalSpeak(dog);  // Bark!
makeAnimalSpeak(cat);  // Meow!
```

8.2.3 How JavaScript Enables Method Overriding and Dynamic Dispatch

JavaScript achieves method overriding and dynamic dispatch through its **prototype chain**:

- When a method is called on an object, JavaScript first looks for that method directly on the object.
- If not found, it walks up the prototype chain, checking parent objects until it finds the method or reaches the end.
- This means subclasses can override methods simply by defining a method with the same name on their own prototype.
- At runtime, the method belonging to the actual object instance is called — enabling polymorphism.

Example:

```
function Animal() {}

Animal.prototype.speak = function() {
  console.log("Animal makes a sound");
};

function Dog() {}
```

```
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Override speak method
Dog.prototype.speak = function() {
  console.log("Bark!");
};

const dog = new Dog();
dog.speak();  // Bark!
```

8.2.4 Summary

- **Method overriding** allows subclasses to replace or extend parent class methods with their own versions.
- **Dynamic dispatch** ensures that when a method is called, the correct overridden method is selected based on the actual object type at runtime.
- JavaScript's prototype-based inheritance naturally supports this behavior, enabling flexible and polymorphic object-oriented designs.

Understanding these concepts is essential for building extensible systems where objects can behave differently while sharing a common interface.

8.3 Practical: Shape Drawing Example with Polymorphism

To illustrate polymorphism in action, let's create a simple example involving shapes. We'll define a base class `Shape` with a generic `draw()` method, then create derived classes `Circle` and `Rectangle` that override `draw()` to provide specific drawing behavior.

8.3.1 Defining the Base and Derived Classes

```
// Base class
class Shape {
  draw() {
    console.log("Drawing a generic shape.");
  }
}

// Derived class: Circle
class Circle extends Shape {
  constructor(radius) {
```

```
    super();
    this.radius = radius;
  }

  draw() {
    console.log(`Drawing a circle with radius ${this.radius}.`);
  }
}

// Derived class: Rectangle
class Rectangle extends Shape {
  constructor(width, height) {
    super();
    this.width = width;
    this.height = height;
  }

  draw() {
    console.log(`Drawing a rectangle ${this.width}x${this.height}.`);
  }
}
```

8.3.2 Using Polymorphism to Draw Shapes

Now, let's create an array of various shapes and call their `draw()` methods in a loop:

```
const shapes = [
  new Circle(10),
  new Rectangle(20, 15),
  new Shape()  // Generic shape
];

shapes.forEach(shape => {
  shape.draw();  // Calls the appropriate draw() based on the actual object type
});
```

Output:

```
Drawing a circle with radius 10.
Drawing a rectangle 20x15.
Drawing a generic shape.
```

Full runnable code:

```
// Base class
class Shape {
  draw() {
    console.log("Drawing a generic shape.");
  }
}
```

```
// Derived class: Circle
class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  draw() {
    console.log(`Drawing a circle with radius ${this.radius}.`);
  }
}

// Derived class: Rectangle
class Rectangle extends Shape {
  constructor(width, height) {
    super();
    this.width = width;
    this.height = height;
  }

  draw() {
    console.log(`Drawing a rectangle ${this.width}x${this.height}.`);
  }
}
const shapes = [
  new Circle(10),
  new Rectangle(20, 15),
  new Shape()  // Generic shape
];

shapes.forEach(shape => {
  shape.draw();  // Calls the appropriate draw() based on the actual object type
});
```

8.3.3 Explanation

- Each class defines a `draw()` method. The derived classes override the base `Shape` method with specific implementations.
- When we iterate over the `shapes` array and call `draw()`, **dynamic dispatch** ensures the correct method runs for each object.
- The loop treats all objects as generic `Shape` instances, yet the behavior varies depending on the actual object — that's polymorphism.
- This approach promotes **flexibility**: new shapes can be added without changing the code that calls `draw()`.
- It also enhances **code reuse** by allowing shared interfaces (`draw()` method) while enabling customized behavior.

8.3.4 Summary

This example shows how polymorphism allows a collection of different objects to be processed uniformly while retaining their unique behaviors. This pattern is powerful for building scalable, maintainable applications that handle diverse object types seamlessly.

Chapter 9.

Mixins and Multiple Inheritance Patterns

1. What Are Mixins?
2. Implementing Mixins in JavaScript
3. Composition over Inheritance
4. Practical: Adding Logging Functionality with Mixins

9 Mixins and Multiple Inheritance Patterns

9.1 What Are Mixins?

In object-oriented programming, **mixins** are a powerful pattern that allows you to add reusable functionality to classes without relying on classical inheritance. Unlike traditional inheritance, where a class inherits from a single parent, mixins enable **sharing behavior across unrelated classes** by "mixing in" methods and properties.

9.1.1 Why JavaScript Doesnt Support Multiple Inheritance

JavaScript's inheritance model is **prototype-based** and only supports **single inheritance**, meaning an object or class can inherit from just one prototype or parent class. Unlike some other languages that allow multiple inheritance (where a class can extend multiple parents), JavaScript avoids this complexity to prevent problems like the "diamond problem," where ambiguities arise about which parent's method to use.

9.1.2 How Mixins Provide a Flexible Alternative

Mixins provide a way to **compose behavior** from multiple sources by copying or merging properties and methods into a target class or object. This technique:

- Allows you to **reuse common functionality** across different classes without forcing them into a rigid inheritance hierarchy.
- Encourages **composition over inheritance**, which is often more flexible and easier to maintain.
- Helps you **avoid deep and complex inheritance trees**, which can become hard to understand and debug.

For example, if multiple classes need logging functionality, instead of creating a shared parent just for logging, you can create a **logging mixin** and apply it to any class that needs it.

9.1.3 When to Use Mixins

Mixins are particularly useful when:

- You want to add **cross-cutting concerns** like logging, event handling, or validation.
- Different classes share **common behavior** but don't fit naturally into a single inheritance hierarchy.

- You want to keep your codebase **modular and maintainable** by reusing small, focused pieces of functionality.

9.1.4 Summary

Mixins are a practical way to enhance JavaScript classes with shared features without the constraints of multiple inheritance. By promoting composition and avoiding deep inheritance chains, mixins help you build more flexible and reusable code. Understanding this pattern is key to mastering advanced object-oriented design in JavaScript.

9.2 Implementing Mixins in JavaScript

Mixins in JavaScript can be implemented in several practical ways, allowing you to **add reusable functionality** to classes or objects without using classical inheritance. Below, we'll explore common techniques, provide examples, and discuss their advantages and potential pitfalls.

9.2.1 Copying Properties onto a Class Prototype

One straightforward approach is to **copy methods and properties directly onto a class's prototype** using `Object.assign()`. This way, all instances of the class gain access to the mixin's functionality.

```
// Define a mixin object with reusable methods
const LoggerMixin = {
  log(message) {
    console.log(`[LOG] ${message}`);
  },
  warn(message) {
    console.log(`[WARN] ${message}`);
  }
};

// Base class
class User {
  constructor(name) {
    this.name = name;
  }
}

// Add mixin methods to User's prototype
Object.assign(User.prototype, LoggerMixin);
```

```
const user = new User("Alice");
user.log("User created");   // Output: [LOG] User created
user.warn("Low disk space"); // Output: [WARN] Low disk space
```

9.2.2 Using Functions to Augment Classes (Mixin Factories)

Another flexible pattern is to create **functions that take a class and return an extended version** of it with added behavior. This approach supports mixin composition.

```
const TimestampMixin = (Base) => class extends Base {
  getTimestamp() {
    return new Date().toISOString();
  }
};

class Document {
  constructor(title) {
    this.title = title;
  }
}

// Apply mixin to Document class
class TimestampedDocument extends TimestampMixin(Document) {}

const doc = new TimestampedDocument("My Doc");
console.log(doc.title);            // My Doc
console.log(doc.getTimestamp());   // Current ISO timestamp
```

9.2.3 Advantages of Mixins

- **Reusable code:** Mixins let you share functionality without forcing class hierarchies.
- **Composition-friendly:** Multiple mixins can be combined to extend classes.
- **Decoupling:** Keeps concerns modular and separate.

9.2.4 Caveats and How to Manage Them

- **Naming conflicts:** If multiple mixins define methods with the same name, they may overwrite each other unpredictably. To avoid this:
  - Use **unique method names** or namespaces.
  - Carefully order mixin applications.
- **State management:** Mixins typically add methods, but managing shared state

requires careful design.

- **Increased complexity:** Overuse of mixins can make code harder to follow if responsibilities are scattered.

9.2.5 Summary

Implementing mixins in JavaScript offers a flexible way to extend classes and objects beyond traditional inheritance. Whether by copying properties onto prototypes or creating higher-order functions to augment classes, mixins promote code reuse and modular design. However, mindful use is essential to avoid conflicts and maintain readability.

9.3 Composition over Inheritance

One of the core principles in modern object-oriented design is to **favor composition over inheritance**. This means building complex objects by assembling smaller, reusable pieces of functionality rather than relying heavily on deep and rigid class hierarchies. Let's explore why this approach matters, how mixins support it, and what benefits it brings.

9.3.1 Why Favor Composition?

Inheritance can sometimes lead to:

- **Deep, complicated hierarchies** that are hard to maintain or understand.
- **Tight coupling** between parent and child classes, making changes difficult.
- **Fragile designs** where modifying one class may unintentionally affect many subclasses.

Composition, on the other hand, encourages you to build objects by **combining focused behaviors**. This keeps code modular, easier to reason about, and promotes reuse across unrelated classes.

9.3.2 How Mixins Enable Composition

Mixins are a natural fit for composition because they let you **"mix in" small, well-defined behaviors** into any class or object. Instead of inheriting everything from a single parent, you pick and choose which capabilities to add, promoting flexibility and separation of concerns.

9.3.3 Conceptual Example: Inheritance vs. Composition

Inheritance example:

```
class Vehicle {
  start() { console.log("Starting vehicle"); }
}

class Car extends Vehicle {
  openTrunk() { console.log("Opening trunk"); }
}

class SportsCar extends Car {
  turboBoost() { console.log("Turbo boost activated"); }
}
```

- Here, `SportsCar` inherits all behaviors from `Car` and `Vehicle`.
- Adding a new feature requires creating a new subclass or modifying existing ones.
- The hierarchy can become complex and rigid.

Composition example with mixins:

```
const Startable = {
  start() { console.log("Starting engine"); }
};

const TrunkOpenable = {
  openTrunk() { console.log("Opening trunk"); }
};

const TurboCharged = {
  turboBoost() { console.log("Turbo boost activated"); }
};

class SportsCar {}
Object.assign(SportsCar.prototype, Startable, TrunkOpenable, TurboCharged);

const myCar = new SportsCar();
myCar.start();        // Starting engine
myCar.openTrunk();    // Opening trunk
myCar.turboBoost();   // Turbo boost activated
```

- Here, the `SportsCar` gains behaviors by composing small mixins.
- You can reuse `Startable` or `TurboCharged` in other classes without inheritance.
- The design remains flexible and modular.

9.3.4 Summary

Favoring composition over inheritance helps you create **more maintainable and adaptable code**. Mixins are a practical tool to support this philosophy by allowing you to compose

objects from focused behaviors, avoiding the pitfalls of complex inheritance trees. This approach promotes **code reuse, clarity, and flexibility** in your JavaScript OOP designs.

9.4 Practical: Adding Logging Functionality with Mixins

In this section, we'll build a **logging mixin** that can be added to any class to provide logging capabilities. This mixin will allow classes to log method calls or state changes **without altering their inheritance**—demonstrating the power and flexibility of mixins in JavaScript.

9.4.1 Step 1: Define the Logging Mixin

We'll create a simple mixin that adds two methods: `log` and `logState`. These methods will output messages to the console, helping track what's happening inside objects.

```
const LoggingMixin = {
  log(message) {
    console.log(`[LOG] ${message}`);
  },
  logState(stateName, stateValue) {
    console.log(`[STATE] ${stateName}:`, stateValue);
  }
};
```

9.4.2 Step 2: Create Some Classes to Extend

Next, we'll define two example classes, `User` and `Product`, which will benefit from logging but don't need to share a common ancestor other than `Object`.

```
class User {
  constructor(name) {
    this.name = name;
  }
  updateName(newName) {
    this.name = newName;
    this.log(`User's name updated to ${newName}`);
  }
}

class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }
```

```
  changePrice(newPrice) {
    this.price = newPrice;
    this.logState('price', newPrice);
  }
}
```

9.4.3 Step 3: Apply the Mixin to Classes

We use `Object.assign` to copy the logging methods into each class's prototype. This adds logging capabilities without modifying the class inheritance or constructor.

```
Object.assign(User.prototype, LoggingMixin);
Object.assign(Product.prototype, LoggingMixin);
```

9.4.4 Step 4: Using the Classes with Logging

Now we create instances and call methods that trigger logging.

```
const user = new User("Alice");
user.updateName("Alicia");
// Output: [LOG] User's name updated to Alicia

const product = new Product("Laptop", 1200);
product.changePrice(1100);
// Output: [STATE] price: 1100
```

Full runnable code:

```
// Step 1: Define the Logging Mixin
const LoggingMixin = {
  log(message) {
    console.log(`[LOG] ${message}`);
  },
  logState(stateName, stateValue) {
    console.log(`[STATE] ${stateName}:`, stateValue);
  }
};

// Step 2: Create Some Classes to Extend
class User {
  constructor(name) {
    this.name = name;
  }

  updateName(newName) {
    this.name = newName;
    this.log(`User's name updated to ${newName}`);
```

```
  }
}

class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  changePrice(newPrice) {
    this.price = newPrice;
    this.logState('price', newPrice);
  }
}

// Step 3: Apply the Mixin to Classes
Object.assign(User.prototype, LoggingMixin);
Object.assign(Product.prototype, LoggingMixin);

// Step 4: Using the Classes with Logging
const user = new User("Alice");
user.updateName("Alicia");
// Output: [LOG] User's name updated to Alicia

const product = new Product("Laptop", 1200);
product.changePrice(1100);
// Output: [STATE] price: 1100
```

9.4.5 Why This Works Well

- **Reusability:** The `LoggingMixin` can be applied to any class without inheritance constraints.
- **Separation of concerns:** Logging logic is kept separate from core business logic.
- **Flexibility:** Easily add or remove logging capabilities without changing the class hierarchy.

9.4.6 Summary

This example demonstrates how mixins can add powerful, reusable behaviors like logging across unrelated classes. By composing classes with mixins, you keep your code modular, maintainable, and flexible—key benefits of modern JavaScript OOP design.

Chapter 10.

The Module Pattern and Encapsulation

1. Why Modules Matter in OOP
2. Classic Module Pattern Using IIFEs
3. ES6 Modules and Classes
4. Practical: Creating a Calculator Module

10 The Module Pattern and Encapsulation

10.1 Why Modules Matter in OOP

Modularity and encapsulation are foundational principles in object-oriented programming (OOP) that help organize code into manageable, reusable, and maintainable units. In JavaScript, **modules** play a vital role in achieving these goals by providing a structured way to separate concerns, limit scope pollution, and protect private data.

10.1.1 The Need for Modularity

As applications grow larger and more complex, managing code without a clear structure becomes increasingly difficult. Without modularity, all code lives in a shared global space, leading to issues such as:

- **Name collisions:** Multiple variables or functions with the same name interfere with each other.
- **Unintended dependencies:** Different parts of the application become tightly coupled, making changes risky and time-consuming.
- **Difficult maintenance:** Understanding and updating code becomes a challenge due to poor organization.

Modules solve these problems by breaking the program into **self-contained units**, each with a well-defined purpose. This separation of concerns means that each module handles a specific task or feature, making it easier to develop, test, and debug independently.

10.1.2 Encapsulation and Private Data

Modules also provide **encapsulation**, which is the practice of hiding internal details and exposing only what is necessary. By encapsulating private data and implementation details, modules:

- **Prevent unintended access or modification** of internal state.
- **Reduce bugs and improve security** by limiting what external code can affect.
- **Promote cleaner APIs** that expose only essential interfaces.

In JavaScript, early techniques such as Immediately Invoked Function Expressions (IIFEs) allowed developers to create private scopes to achieve encapsulation. More modern systems, like ES6 modules, provide native syntax to define and import/export modules cleanly.

10.1.3 Benefits in Large Applications and Team Environments

For large projects and teams, modularity is critical:

- **Clear boundaries** between modules help teams work independently on different features.
- **Easier code sharing and reuse** across projects.
- **Improved scalability** by allowing incremental development and integration.
- **Better code organization** reduces technical debt and improves onboarding of new developers.

10.1.4 Evolution of JavaScript Module Systems

JavaScript's module capabilities have evolved significantly:

- **Global scripts:** Early JavaScript had no module system, leading to global scope pollution.
- **IIFE-based modules:** Developers created modules by wrapping code in functions to limit scope.
- **CommonJS and AMD:** Introduced for server-side and asynchronous loading in browsers.
- **ES6 Modules:** The modern standard providing static `import` and `export` syntax with built-in support in most environments.

Today, ES6 modules represent the best practice for modular JavaScript development, supporting powerful encapsulation and composability.

10.2 Classic Module Pattern Using IIFEs

Before the introduction of ES6 modules, JavaScript developers relied on clever patterns to achieve modularity and encapsulation. One of the most popular and effective techniques was the **Immediately Invoked Function Expression (IIFE)**. This pattern uses a self-executing function to create a **private scope**, isolating variables and functions from the global environment, thus simulating a module.

10.2.1 What is an IIFE?

An IIFE is a function that is defined and executed immediately. Its syntax typically looks like this:

```
(function() {
  // Code here runs immediately and is private
})();
```

By wrapping code inside this function, you create a scope that is **not accessible** from outside. Variables declared inside the IIFE cannot be accidentally modified or accessed from the global scope, helping to prevent naming conflicts and preserve privacy.

10.2.2 Using IIFEs to Create Modules

The **Module Pattern** builds on IIFEs by returning an object that exposes a **public API** while keeping internal data and helper functions hidden. This leverages **closures**—functions retaining access to their lexical environment—even after the IIFE has executed.

10.2.3 Example: A Simple Counter Module

```
const Counter = (function() {
  // Private variable
  let count = 0;

  // Private helper function
  function logCount() {
    console.log(`Current count: ${count}`);
  }

  // Public API exposed by the module
  return {
    increment() {
      count++;
      logCount();
    },
    decrement() {
      count--;
      logCount();
    },
    getCount() {
      return count;
    }
  };
})();

// Usage:
Counter.increment();  // Current count: 1
Counter.increment();  // Current count: 2
console.log(Counter.getCount()); // 2
Counter.decrement();  // Current count: 1
```

10.2.4 Explanation:

- The **IIFE** creates a private scope for `count` and `logCount`.
- The returned object exposes only the methods `increment`, `decrement`, and `getCount`.
- External code cannot access or modify `count` directly, ensuring **data encapsulation**.
- The exposed methods form a controlled interface to interact with the internal state.

10.2.5 Benefits and Limitations

- **Benefits:**
  - Encapsulates private data and implementation details.
  - Avoids polluting the global scope.
  - Creates clear public APIs with controlled access.
- **Limitations:**
  - No native support for module dependencies (managed manually).
  - Verbose syntax compared to ES6 modules.
  - Less intuitive for newcomers.

The IIFE-based module pattern was a critical step in JavaScript's evolution towards better modularity and encapsulation. It allowed developers to write maintainable code long before native module support became widespread. In the next section, we will explore how ES6 modules and classes simplify these patterns with standardized syntax.

10.3 ES6 Modules and Classes

With the advent of ES6 (ECMAScript 2015), JavaScript introduced a **native module system** designed to improve code organization, encapsulation, and dependency management. This modern module system addresses many limitations of earlier patterns like IIFEs and script concatenation.

10.3.1 ES6 Modules: `export` and `import`

ES6 modules allow you to define pieces of functionality—variables, functions, classes—in one file and **export** them so other files can **import** and use them. This explicit syntax makes dependencies clear and facilitates better tooling and optimization.

Exporting

There are two main ways to export from a module:

- **Named exports:**

```
export const PI = 3.14159;

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

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

- **Default export:** (only one per module)

```
export default class Calculator {
  add(a, b) {
    return a + b;
  }
}
```

Importing

You can import named exports or default exports into other modules:

```
import { PI, greet } from './utils.js';

import Calculator from './Calculator.js';

console.log(PI); // 3.14159
greet('Alice');  // Hello, Alice!

const calc = new Calculator();
console.log(calc.add(2, 3)); // 5
```

10.3.2 Static Structure and Advantages

- **Static analysis:** Since `import` and `export` statements are static, JavaScript engines and build tools can analyze dependencies before code runs. This enables optimizations such as **tree shaking**, where unused code is eliminated during bundling, resulting in smaller and faster applications.
- **Scope isolation:** Modules have their own scope by default, preventing accidental globals and making encapsulation easier.
- **Clear dependencies:** Import statements explicitly declare a module's dependencies

at the top, improving code readability and maintainability.

10.3.3 Classes Within ES6 Modules

Classes integrate naturally into ES6 modules as first-class exports. This pairing encourages writing reusable, encapsulated components:

- You can define a class inside a module and export it for use elsewhere.
- Classes encapsulate data and behavior, while modules organize those classes and other related functionality into logical units.
- This combination supports **separation of concerns** and **modular architecture**, crucial for large-scale applications.

10.3.4 Differences from Classic Patterns

Compared to classic module patterns like IIFEs:

- ES6 modules are **language-native**, standardized, and supported directly by modern browsers and JavaScript runtimes.
- They avoid manual export objects and IIFE wrappers.
- Provide **built-in dependency resolution**, eliminating the need for global namespace management.
- Support asynchronous loading and can be combined with modern build tools seamlessly.

10.3.5 Supporting Modern Development Workflows

ES6 modules have become the backbone of modern JavaScript development:

- They enable advanced bundlers like Webpack, Rollup, and Parcel to optimize and bundle code efficiently.
- Facilitate code splitting and lazy loading, improving application performance.
- Enhance maintainability in team environments by promoting modular, reusable code.

In summary, ES6 modules and classes together provide a powerful, clean, and scalable approach to building JavaScript applications with clear boundaries and encapsulation, significantly improving on older patterns and setting the foundation for modern web development. The next section will demonstrate these concepts in action by creating a calculator module.

10.4 Practical: Creating a Calculator Module

To reinforce the concepts of encapsulation and modular design, let's build a simple calculator module using two different approaches:

1. **Classic Module Pattern (IIFE)**
2. **ES6 Module with Class Syntax**

Both implementations will expose a clean API for performing basic arithmetic operations—addition, subtraction, multiplication, and division—while hiding internal implementation details.

10.4.1 A. Classic Module Pattern Using IIFE

Before ES6, JavaScript developers used Immediately Invoked Function Expressions (IIFEs) to create modules with private scope.

```
// calculatorModule.js (Classic IIFE Module)
const Calculator = (function () {
  // Private helper (not exposed)
  function validateInput(a, b) {
    if (typeof a !== 'number' || typeof b !== 'number') {
      throw new Error('Inputs must be numbers');
    }
  }

  // Public API
  return {
    add(a, b) {
      validateInput(a, b);
      return a + b;
    },
    subtract(a, b) {
      validateInput(a, b);
      return a - b;
    },
    multiply(a, b) {
      validateInput(a, b);
      return a * b;
    },
    divide(a, b) {
      validateInput(a, b);
      if (b === 0) throw new Error('Division by zero');
      return a / b;
    },
  };
})();
```

Usage

```
console.log(Calculator.add(5, 3));        // 8
console.log(Calculator.divide(10, 2));    // 5
```

Highlights:

- Encapsulation is achieved using function scope.
- Only the returned object (`add`, `subtract`, etc.) is accessible.
- Internal logic like `validateInput` remains private.

10.4.2 B. ES6 Module Using a Class

With ES6, we can define modules using the `export`/`import` syntax and encapsulate functionality in classes.

```
// Calculator.js (ES6 Module)
export default class Calculator {
  static validateInput(a, b) {
    if (typeof a !== 'number' || typeof b !== 'number') {
      throw new Error('Inputs must be numbers');
    }
  }

  add(a, b) {
    Calculator.validateInput(a, b);
    return a + b;
  }

  subtract(a, b) {
    Calculator.validateInput(a, b);
    return a - b;
  }

  multiply(a, b) {
    Calculator.validateInput(a, b);
    return a * b;
  }

  divide(a, b) {
    Calculator.validateInput(a, b);
    if (b === 0) throw new Error('Division by zero');
    return a / b;
  }
}
```

Usage (in another file)

```
// main.js
import Calculator from './Calculator.js';

const calc = new Calculator();
```

```
console.log(calc.add(4, 2));        // 6
console.log(calc.multiply(3, 7));   // 21
```

Highlights:

- Public methods are encapsulated within a class.
- `validateInput` is a static method, scoped to the class and not tied to instances.
- The module file exposes the class while hiding internal logic.

10.4.3 Comparison and Use Cases

Feature	Classic Module (IIFE)	ES6 Module + Class
Scope isolation	Function closure	Module + class scope
Private members	Via closure	Via class or private fields
Reusability	Singleton only	Multiple instances possible
Tooling support	Limited	Native tree shaking, linting
Modern compatibility	Universal	Modern environments only

10.4.4 Conclusion

Both patterns demonstrate encapsulation and clean API design. The **IIFE module** is suitable for simple, singleton-like utilities, while the **ES6 class module** is ideal for reusable, scalable components in modern applications.

This exercise illustrates how thoughtful modular design leads to maintainable, well-organized code—an essential skill in object-oriented JavaScript development.

Chapter 11.

Symbols and Meta-Programming

1. Using Symbols for Private and Unique Properties
2. Property Descriptors and `Object.defineProperty`
3. Proxy Objects and Traps
4. Practical: Creating a Validation Proxy for Objects

11 Symbols and Meta-Programming

11.1 Using Symbols for Private and Unique Properties

In JavaScript, **Symbols** are a special, primitive data type introduced in ES6. They are used to create unique, immutable identifiers that can serve as keys for object properties. Unlike strings or numbers, each Symbol is guaranteed to be **unique**, even if it has the same description. This makes Symbols ideal for creating properties that are **hidden from most operations** and avoid naming conflicts—an important part of achieving encapsulation and safe extension of objects.

11.1.1 What Are Symbols?

A Symbol is created by calling the `Symbol()` function:

```
const id = Symbol('id');
```

Here, `'id'` is just a description used for debugging—it does not affect the uniqueness of the Symbol. Every time you call `Symbol()`, a new, unique Symbol is returned:

```
const sym1 = Symbol('user');
const sym2 = Symbol('user');

console.log(sym1 === sym2); // false
```

Despite having the same description, `sym1` and `sym2` are not equal. This ensures that property keys using Symbols will never accidentally overwrite each other, which is useful in large codebases or when dealing with third-party code.

11.1.2 Using Symbols as Object Property Keys

Symbols can be used as keys in object literals or with bracket notation:

```
const secretKey = Symbol('secret');

const user = {
  name: 'Alice',
  age: 30,
  [secretKey]: 'encrypted-password',
};

console.log(user.name); // Alice
console.log(user[secretKey]); // encrypted-password
```

Here, `secretKey` is a Symbol that uniquely identifies a property. This property is **not accessible via standard object iteration** methods:

```
for (let key in user) {
  console.log(key); // Outputs: name, age (not the symbol)
}

console.log(Object.keys(user)); // ['name', 'age']
console.log(Object.getOwnPropertyNames(user)); // ['name', 'age']
console.log(Object.getOwnPropertySymbols(user)); // [Symbol(secret)]
```

This behavior makes Symbol-keyed properties **semi-private**—they are not truly private (like private fields with `#`), but they're hidden from most casual inspection and enumeration.

11.1.3 Preventing Name Collisions

In scenarios where multiple scripts or modules might define properties on the same object (e.g., extending built-in prototypes), using Symbols ensures that your properties won't clash:

```
const toJSONSymbol = Symbol('toJSON');

class SecureData {
  constructor(data) {
    this.data = data;
    this[toJSONSymbol] = function () {
      return '[REDACTED]';
    };
  }

  toJSON() {
    return this[toJSONSymbol]();
  }
}

const record = new SecureData('top-secret');
console.log(JSON.stringify(record)); // "[REDACTED]"
```

In this example, the internal `toJSONSymbol` method avoids conflicting with any existing `toJSON` methods elsewhere.

11.1.4 Summary

- **Symbols** are primitive values that serve as **unique, non-enumerable property keys**.
- They provide a mechanism for **avoiding property name collisions** and for creating **semi-private object members**.

- Symbols help encapsulate internal implementation details without polluting or risking clashes in the public API surface of your objects.

Symbols are a powerful tool in your JavaScript OOP toolbox—especially when writing extensible, modular, or secure code. In the next sections, we'll explore more meta-programming features like property descriptors and Proxies that pair well with Symbols for advanced object control.

11.2 Property Descriptors and `Object.defineProperty`

JavaScript objects are flexible and dynamic, but when you need **precise control over how properties behave**, property descriptors come into play. Property descriptors are objects that define the characteristics of object properties—such as whether a property is writable, enumerable, or configurable.

11.2.1 What Are Property Descriptors?

When you define a property in an object using standard syntax:

```
const user = {
  name: "Alice",
};
```

JavaScript automatically assigns default settings for that property:

- **writable**: `true` (you can change the value)
- **enumerable**: `true` (shows up in loops and `Object.keys`)
- **configurable**: `true` (can be deleted or modified)

You can view a property's descriptor using `Object.getOwnPropertyDescriptor()`:

```
console.log(Object.getOwnPropertyDescriptor(user, "name"));
/*
{
  value: "Alice",
  writable: true,
  enumerable: true,
  configurable: true
}
*/
```

11.2.2 Using `Object.defineProperty()`

The `Object.defineProperty()` method lets you define or modify a property's descriptor explicitly:

```
const person = {};

Object.defineProperty(person, "ssn", {
  value: "123-45-6789",
  writable: false,
  enumerable: false,
  configurable: false
});
```

Now `ssn` is:

- **non-writable**: its value can't be changed
- **non-enumerable**: won't appear in loops or `Object.keys()`
- **non-configurable**: can't be deleted or redefined

```
console.log(person.ssn);            // "123-45-6789"
person.ssn = "000-00-0000";         // Silently fails (or throws in strict mode)
console.log(person.ssn);            // Still "123-45-6789"
console.log(Object.keys(person));   // []
delete person.ssn;                  // Fails silently
```

This approach is especially useful for protecting sensitive data or hiding internal details.

11.2.3 Getters and Setters with Descriptors

Property descriptors can also define **computed properties** using `get` and `set`:

```
const account = {
  firstName: "Jane",
  lastName: "Doe"
};

Object.defineProperty(account, "fullName", {
  get() {
    return `${this.firstName} ${this.lastName}`;
  },
  set(value) {
    const [first, last] = value.split(" ");
    this.firstName = first;
    this.lastName = last;
  },
  enumerable: true,
  configurable: true
});

console.log(account.fullName); // "Jane Doe"
```

```
account.fullName = "Mary Smith";
console.log(account.firstName); // "Mary"
console.log(account.lastName);  // "Smith"
```

This defines a **virtual property**—it behaves like a regular property but is dynamically calculated and stored elsewhere.

11.2.4 Summary of Descriptor Flags

Descriptor Property	Description
`value`	The property's actual value (for data properties)
`writable`	If `true`, the value can be changed
`enumerable`	If `true`, property appears in loops and `Object.keys()`
`configurable`	If `true`, the property can be deleted or reconfigured
`get`	Function called when the property is read
`set`	Function called when the property is written to

Note: A property cannot have both `value` and `get/set`. You must choose between a **data property** or an **accessor property**.

11.2.5 Why Use Descriptors?

Property descriptors allow you to:

- Create **read-only** or **write-once** properties
- **Hide properties** from enumeration
- Protect APIs by marking properties as **non-configurable**
- Build **computed values** through `get/set`

Together with tools like `Object.freeze()` or `Proxy`, property descriptors offer **fine-grained control**—making them a powerful feature for secure, encapsulated OOP design in JavaScript.

11.3 Proxy Objects and Traps

JavaScript's `Proxy` object is a powerful **meta-programming** tool introduced in ES6 that allows developers to intercept and redefine low-level operations on objects. A proxy acts as a **wrapper** around an object, and instead of interacting with the original object directly, operations are routed through the proxy—where they can be customized or blocked. This

mechanism is made possible by **traps**—functions that "trap" operations like reading, writing, or deleting properties.

11.3.1 What Is a Proxy?

A proxy is created using the `Proxy` constructor:

```
const proxy = new Proxy(target, handler);
```

- **`target`** is the original object being wrapped.
- **`handler`** is an object with **trap methods** that define custom behavior for various operations on the target.

11.3.2 Common Traps and Their Uses

Here are a few commonly used traps and what they intercept:

Trap	Intercepts…
`get`	Reading a property value
`set`	Assigning a property value
`has`	Using the `in` operator
`deleteProperty`	Using the `delete` keyword
`ownKeys`	Getting a list of keys
`apply`	Calling a function (function proxies only)

Example: Logging Property Access

```
const user = {
  name: "Alice",
  role: "admin"
};

const loggingHandler = {
  get(target, property) {
    console.log(`Accessed property: ${property}`);
    return target[property];
  }
};

const userProxy = new Proxy(user, loggingHandler);

console.log(userProxy.name); // Logs: Accessed property: name
```

In this example, every time a property is read, the `get` trap logs it. This is useful for debugging or monitoring sensitive data access.

11.3.3 Example: Validation on Property Set

```
const validator = {
  set(target, property, value) {
    if (property === "age" && typeof value !== "number") {
      throw new TypeError("Age must be a number");
    }
    target[property] = value;
    return true;
  }
};

const person = new Proxy({}, validator);

person.age = 30;       // OK
person.name = "Tom";   // OK
person.age = "old";    // Throws: TypeError: Age must be a number
```

Here, the `set` trap validates property assignments before they happen. This enables **runtime enforcement** of data constraints without having to define setters for every property.

11.3.4 Use Cases for Proxies

Proxies are particularly useful in:

1. **Validation** Enforce rules for object structure or data types dynamically.
2. **Access Control / Encapsulation** Hide or protect sensitive properties from external access.
3. **Debugging and Logging** Track reads/writes to object properties during runtime.
4. **Virtualization and Lazy Evaluation** Simulate non-existent properties or load data on demand.
5. **Reactive Frameworks** Libraries like Vue.js use proxies to detect changes in data and update the DOM reactively.

11.3.5 Conceptual Power of Traps

Traps decouple the internal behavior of an object from its external interface. They allow objects to behave like **smart components** that monitor, reject, transform, or virtualize operations as needed. With traps, the behavior of an object is **not fixed**—it becomes programmable at the meta-level.

11.3.6 Important Caveats

- Proxies only affect operations **on the proxy object**, not directly on the target.
- Some native operations may not fully respect proxies (especially in older environments).
- Overusing proxies can make code harder to debug and maintain, so they should be used thoughtfully.

11.3.7 Summary

Proxies give JavaScript developers the ability to **intercept and customize nearly every interaction with an object**. Through traps like `get` and `set`, you can implement behaviors like validation, logging, lazy-loading, or even virtual properties. As a meta-programming tool, proxies unlock a high level of control and flexibility, making them ideal for complex, dynamic applications.

In the next section, we'll apply this knowledge to create a validation proxy that ensures objects meet specific criteria when properties are set or accessed.

11.4 Practical: Creating a Validation Proxy for Objects

JavaScript proxies are powerful tools for enforcing data validation at runtime. By intercepting property assignments using the `set` trap, you can ensure that only valid values are accepted, without having to define custom setters for each property.

In this section, we'll walk through building a `Proxy` that validates the properties of a `UserProfile` object. We'll ensure, for example, that the `age` is a number within a valid range and that the `email` contains an `@` symbol.

11.4.1 Step-by-Step Example: User Profile Validation

Let's start by defining a target object and a validation handler that enforces rules when properties are assigned.

```
// Step 1: Define a plain target object
const userProfile = {
  name: "",
  age: 0,
  email: ""
};
```

11.4.2 Step 2: Create a Validation Handler with a `set` Trap

The `set` trap intercepts all property assignments and allows us to add validation logic before allowing the assignment.

```
const validationHandler = {
  set(target, property, value) {
    switch (property) {
      case "name":
        if (typeof value !== "string" || value.trim() === "") {
          throw new Error("Name must be a non-empty string.");
        }
        break;
      case "age":
        if (typeof value !== "number" || value < 0 || value > 120) {
          throw new Error("Age must be a number between 0 and 120.");
        }
        break;
      case "email":
        if (typeof value !== "string" || !value.includes("@")) {
          throw new Error("Email must be a valid email address.");
        }
        break;
      default:
        console.warn(`No validation rule for property: ${property}`);
    }

    // If validation passed, assign the value to the target
    target[property] = value;
    return true; // Must return true to indicate success
  }
};
```

11.4.3 Step 3: Create the Proxy

Now we wrap the target object with our validation handler using the `Proxy` constructor.

```
const validatedUser = new Proxy(userProfile, validationHandler);
```

11.4.4 Step 4: Try Assigning Values

Let's test the proxy by assigning various values.

```
try {
  validatedUser.name = "Alice";         // OK
  validatedUser.age = 30;               // OK
  validatedUser.email = "alice@web.com"; // OK

  console.log("User profile set successfully:", validatedUser);
} catch (err) {
  console.error("Validation error:", err.message);
}

try {
  validatedUser.age = "old"; // Error: Age must be a number between 0 and 120.
} catch (err) {
  console.error("Validation error:", err.message);
}

try {
  validatedUser.email = "invalidemail"; // Error: Email must be a valid email address.
} catch (err) {
  console.error("Validation error:", err.message);
}
```

Full runnable code:

```
// Step 1: Define a plain target object
const userProfile = {
  name: "",
  age: 0,
  email: ""
};

// Step 2: Create a Validation Handler with a `set` Trap
const validationHandler = {
  set(target, property, value) {
    switch (property) {
      case "name":
        if (typeof value !== "string" || value.trim() === "") {
          throw new Error("Name must be a non-empty string.");
        }
        break;
      case "age":
        if (typeof value !== "number" || value < 0 || value > 120) {
          throw new Error("Age must be a number between 0 and 120.");
        }
        break;
      case "email":
```

```
        if (typeof value !== "string" || !value.includes("@")) {
          throw new Error("Email must be a valid email address.");
        }
        break;
      default:
        console.warn(`No validation rule for property: ${property}`);
    }

    // If validation passed, assign the value to the target
    target[property] = value;
    return true; // Must return true to indicate success
  }
};

// Step 3: Create the Proxy
const validatedUser = new Proxy(userProfile, validationHandler);

// Step 4: Try Assigning Values
try {
  validatedUser.name = "Alice";        // OK
  validatedUser.age = 30;              // OK
  validatedUser.email = "alice@web.com"; // OK

  console.log("User profile set successfully:", validatedUser);
} catch (err) {
  console.error("Validation error:", err.message);
}

try {
  validatedUser.age = "old"; // Error: Age must be a number between 0 and 120.
} catch (err) {
  console.error("Validation message:", err.message);
}

try {
  validatedUser.email = "invalidemail"; // Error: Email must be a valid email address.
} catch (err) {
  console.error("Validation message:", err.message);
}
```

11.4.5 Why Use a Validation Proxy?

Using a validation proxy centralizes data integrity checks in one reusable place. Benefits include:

- **Decoupled logic**: Validation logic is not scattered across the application.
- **Flexibility**: Easily add or remove rules without changing the base object.
- **Security**: Prevents silent data corruption or invalid states.

11.4.6 Tips and Caveats

- Always return `true` from the `set` trap to indicate success; otherwise, JavaScript will throw a `TypeError`.
- Use `Reflect.set(target, property, value)` to delegate to the default behavior, especially in more complex proxies.
- You can apply similar validation logic for nested objects using recursive proxies.

11.4.7 Summary

In this practical example, we saw how to use a JavaScript `Proxy` to **validate property assignments dynamically**. By customizing the `set` trap, we enforced type checking and format rules, throwing meaningful errors when invalid data was detected. Proxies like this are powerful tools for **runtime validation**, **secure programming**, and **dynamic behavior**, especially in larger, user-driven applications.

Chapter 12.

Object Immutability and Functional OOP

1. Immutability Concepts
2. Using `Object.freeze` and `Object.seal`
3. Functional Patterns in OOP
4. Practical: Immutable Data Structures Example

12 Object Immutability and Functional OOP

12.1 Immutability Concepts

Immutability is the principle of creating objects whose state **cannot be changed after they are created**. Instead of modifying an existing object, you create a new one with the desired changes. This concept is a cornerstone of functional programming and is becoming increasingly important in modern object-oriented programming (OOP), especially in large-scale and concurrent applications.

12.1.1 What Is Immutability?

An **immutable object** is one whose properties and structure do not change after it is created. For example, if you create an immutable user object with a name and age, any change to that data requires creating a **new object** rather than altering the original.

By contrast, **mutable objects** can have their properties changed at any time, which is how most JavaScript objects behave by default.

```
// Mutable object example
const user = { name: "Alice", age: 30 };
user.age = 31; // Mutation

// Immutable pattern (new object)
const newUser = { ...user, age: 31 }; // No change to original
```

12.1.2 Why Immutability Matters

Predictability and Debugging

Immutable objects reduce bugs caused by unintended side effects. When you know an object won't change, it's easier to reason about what your code is doing at any given moment.

Safer State Management

In UI frameworks like React, immutability helps track changes efficiently. Since previous state objects are never modified, it's easy to compare versions or undo changes by reverting to an earlier object.

Concurrency and Parallelism

Immutable data structures are inherently thread-safe because they can be shared across asynchronous operations without fear of race conditions. This is especially helpful in environments like web workers or server-side JavaScript (e.g., Node.js).

Functional Composition

In functional programming, functions avoid side effects. Immutability aligns perfectly with this principle, as data is passed and transformed through pure functions rather than mutated in place.

12.1.3 Mutable vs. Immutable: A Conceptual Example

```
// Mutable approach
let profile = { name: "Bob", role: "user" };
function promoteToAdmin(user) {
  user.role = "admin"; // Mutates the original object
  return user;
}

// Immutable approach
function promoteToAdminImmutable(user) {
  return { ...user, role: "admin" }; // Returns a new object
}
```

With the immutable version, the original `user` object remains untouched, making the system more predictable and testable.

12.1.4 Summary

Immutability is a key concept for writing **robust, maintainable, and side-effect-free code**. It makes debugging easier, supports safe concurrent programming, and fosters good architectural patterns. As JavaScript applications grow more complex—especially in reactive UI development and server-side logic—**embracing immutability** becomes an essential best practice for scalable, clean code.

In the next sections, we'll explore how to enforce immutability in JavaScript using built-in tools like `Object.freeze` and patterns from functional programming.

12.2 Using `Object.freeze` and `Object.seal`

JavaScript provides two built-in methods—`Object.freeze()` and `Object.seal()`—to help control and restrict changes to objects. These tools are essential for implementing varying degrees of immutability and protecting your data structures from unintended mutations, especially in large or shared codebases.

12.2.1 `Object.freeze()`: Making an Object Fully Immutable

`Object.freeze()` **completely locks down** an object. After freezing:

- No new properties can be added.
- Existing properties cannot be removed.
- Existing properties cannot be reconfigured (e.g., made writable or enumerable).
- The values of existing properties **cannot be changed**.

Example:

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

Object.freeze(user);

user.age = 35;          // Fails silently or throws in strict mode
user.email = "a@b.com"; // Fails silently
delete user.name;       // Fails silently

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

Notes:

- `Object.freeze()` is **shallow**—it only affects the top-level properties. If the object contains nested objects or arrays, they are still mutable unless you recursively freeze them.

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

Object.freeze(account);
account.user.name = "Charlie"; // Allowed! user object is not frozen
```

To deep-freeze an object, you must recursively freeze every nested object.

12.2.2 `Object.seal()`: Preventing Structural Changes

`Object.seal()` is less strict than `Object.freeze()`. When you seal an object:

- You **cannot add or remove** properties.
- You **can still modify the values** of existing properties.
- Property descriptors like `writable` remain unchanged unless modified manually.

Example:

```
const settings = {
  theme: "dark",
  notifications: true
};

Object.seal(settings);

settings.theme = "light";       // Allowed
settings.language = "English";  // Not allowed
delete settings.notifications;  // Not allowed

console.log(JSON.stringify(settings)); // { theme: "light", notifications: true }
```

When to Use `Object.seal()`:

Use sealing when you want to prevent structural changes (adding/removing keys) but still allow data updates. It's a good fit for partially locked configurations where you want to preserve the schema.

12.2.3 Comparing `freeze()` and `seal()`

Feature	`Object.freeze()`	`Object.seal()`
Add new properties	NO Not allowed	NO Not allowed
Remove existing properties	NO Not allowed	NO Not allowed
Modify existing property values	NO Not allowed	YES Allowed
Change property descriptors	NO Not allowed	NO Not allowed
Shallow or deep	Shallow only	Shallow only

12.2.4 Common Use Cases

- `Object.freeze()`
  - Constants or configuration objects that should not change at runtime.
  - Safeguarding critical data from mutation in shared environments.
  - Functional programming patterns requiring immutability.
- `Object.seal()`
  - API objects where structure is fixed but values may change.
  - Settings or options objects with a controlled schema.
  - Intermediate data protection in libraries or frameworks.

12.2.5 Summary

Both `Object.freeze()` and `Object.seal()` are powerful tools for enforcing data integrity in JavaScript. Use `freeze` when you want total immutability, and `seal` when you want to maintain the object's structure but still allow updates to property values. Understanding and applying these methods wisely leads to more predictable and robust object-oriented code—especially as your application scales.

In the next section, we'll explore how functional programming principles intersect with OOP in JavaScript to further enhance code clarity and reliability.

12.3 Functional Patterns in OOP

Modern JavaScript development often blends **object-oriented programming (OOP)** and **functional programming (FP)** to achieve cleaner, more reliable, and maintainable code. While OOP focuses on organizing code around objects and behavior, FP emphasizes **immutability**, **pure functions**, and **function composition**.

By integrating functional principles into object-oriented design, you can enjoy the benefits of both paradigms: the structure and encapsulation of OOP with the predictability and reusability of FP.

12.3.1 Core Functional Concepts Integrated into OOP

Immutability

Immutability means avoiding changes to existing data. Instead of modifying an object's state, you create and return new versions of the object.

Why it matters: Immutable data structures help prevent unintended side effects, making your application easier to debug, reason about, and test.

Example:

```
class Counter {
  constructor(count = 0) {
    this.count = count;
  }

  increment() {
    // Returns a new Counter instance instead of modifying this one
    return new Counter(this.count + 1);
  }
}
```

```
const c1 = new Counter();
const c2 = c1.increment();

console.log(c1.count); // 0
console.log(c2.count); // 1
```

Here, `increment()` is a **pure function** — it doesn't mutate the original object and always returns the same output for the same input.

Pure Functions

A **pure function** is one that:

- Depends only on its input.
- Has no side effects (e.g., it doesn't change global variables or mutate inputs).

In OOP, you can apply pure functions within class methods or as standalone utilities to promote predictable behavior.

Example:

```
function calculateTotal(price, taxRate) {
  return price + price * taxRate;
}

// Usage inside a class:
class Invoice {
  constructor(items) {
    this.items = items;
  }

  getTotal() {
    return this.items.reduce((sum, item) => sum + calculateTotal(item.price, item.tax), 0);
  }
}
```

Function Composition

Composition means combining smaller functions to build more complex behavior. In OOP, composition can be used to delegate behavior to smaller, independent functions or classes, rather than relying on deep inheritance.

Example:

```
const withTimestamp = obj => ({
  ...obj,
  createdAt: new Date()
});

const withID = obj => ({
  ...obj,
  id: Math.random().toString(36).slice(2)
});
```

```
const createUser = name =>
  withID(withTimestamp({ name }));

const user = createUser("Alice");

console.log(JSON.stringify(user,null,2));
// Output: { name: 'Alice', createdAt: ..., id: 'abc123' }
```

In this example, small, pure functions are composed to enrich objects — a functional alternative to inheritance-based solutions.

12.3.2 Benefits of Combining FP with OOP

Functional Pattern	OOP Benefit
Immutability	Avoids shared state bugs; improves concurrency
Pure Functions	Easier to test and debug; promotes reusability
Composition	Reduces reliance on deep inheritance; improves modularity

12.3.3 Best Practices

- Use **object constructors or classes** to structure and encapsulate behavior.
- Prefer **methods that return new objects** over mutating methods.
- Extract **pure utility functions** where possible.
- Combine **composition** with OOP via mixins or higher-order functions.

12.3.4 Summary

Blending functional programming techniques into your object-oriented JavaScript code brings the best of both worlds. You gain the clarity, safety, and testability of FP without losing the organizational strengths of OOP. This hybrid approach is especially powerful in modern JavaScript development, where maintainability and modularity are critical.

Next, we'll apply these principles in a practical example by creating immutable data structures in action.

12.4 Practical: Immutable Data Structures Example

In this section, we'll explore a hands-on example of working with **immutable data structures** in JavaScript — an essential practice for predictable, safe, and bug-resistant applications.

Immutability means we **never change an object directly**. Instead, we **create and return a new object** each time we want to make an update. This is a core idea in functional programming and increasingly important in object-oriented designs that value clean state transitions.

Let's build a simple **immutable list** of tasks (a to-do list) to illustrate this concept.

12.4.1 Step 1: Define an Immutable List Class

```
class ImmutableTaskList {
  constructor(tasks = []) {
    // Freeze the internal array to prevent mutation
    this._tasks = Object.freeze([...tasks]);
  }

  addTask(task) {
    const newTasks = [...this._tasks, task];
    return new ImmutableTaskList(newTasks);
  }

  removeTask(index) {
    const newTasks = this._tasks.filter((_, i) => i !== index);
    return new ImmutableTaskList(newTasks);
  }

  getTasks() {
    // Return a shallow copy to prevent external mutation
    return [...this._tasks];
  }
}
```

12.4.2 Explanation

- The internal array is frozen using `Object.freeze()` to prevent mutation.
- The `addTask()` and `removeTask()` methods return new instances rather than modifying the existing list.
- `getTasks()` exposes the task list safely without risking mutation.

12.4.3 Step 2: Using the ImmutableTaskList

```
const list1 = new ImmutableTaskList();
const list2 = list1.addTask("Learn JavaScript");
const list3 = list2.addTask("Practice coding");
const list4 = list3.removeTask(0);

console.log("List 1:", list1.getTasks()); // []
console.log("List 2:", list2.getTasks()); // ["Learn JavaScript"]
console.log("List 3:", list3.getTasks()); // ["Learn JavaScript", "Practice coding"]
console.log("List 4:", list4.getTasks()); // ["Practice coding"]
```

Full runnable code:

```
class ImmutableTaskList {
  constructor(tasks = []) {
    // Freeze the internal array to prevent mutation
    this._tasks = Object.freeze([...tasks]);
  }

  addTask(task) {
    const newTasks = [...this._tasks, task];
    return new ImmutableTaskList(newTasks);
  }

  removeTask(index) {
    const newTasks = this._tasks.filter((_, i) => i !== index);
    return new ImmutableTaskList(newTasks);
  }

  getTasks() {
    // Return a shallow copy to prevent external mutation
    return [...this._tasks];
  }
}

// usage

const list1 = new ImmutableTaskList();
const list2 = list1.addTask("Learn JavaScript");
const list3 = list2.addTask("Practice coding");
const list4 = list3.removeTask(0);

console.log("List 1:", list1.getTasks()); // []
console.log("List 2:", list2.getTasks()); // ["Learn JavaScript"]
console.log("List 3:", list3.getTasks()); // ["Learn JavaScript", "Practice coding"]
console.log("List 4:", list4.getTasks()); // ["Practice coding"]
```

12.4.4 Benefits in Action

- **No side effects**: Each list instance remains unchanged after creation.

- **Time travel**: You can revisit any previous state (e.g., list2 or list3) at any time.
- **Easier debugging**: Knowing that data doesn't change unless explicitly recreated makes bugs easier to locate and fix.
- **Safer concurrency**: Immutable objects eliminate race conditions caused by shared mutable state.

12.4.5 Bonus: Immutable Record Pattern

We can also build immutable record-like objects:

```
class ImmutableUser {
  constructor({ name, email }) {
    this._data = Object.freeze({ name, email });
  }

  updateName(newName) {
    return new ImmutableUser({ ...this._data, name: newName });
  }

  getData() {
    return { ...this._data };
  }
}

const user1 = new ImmutableUser({ name: "Alice", email: "alice@example.com" });
const user2 = user1.updateName("Alicia");

console.log(JSON.stringify(user1.getData(),null,2)); // { name: "Alice", email: "alice@example.com" }
console.log(JSON.stringify(user2.getData(),null,2)); // { name: "Alicia", email: "alice@example.com" }
```

12.4.6 Summary

Implementing immutability in JavaScript means building data structures that do not change in place. Instead, you return new objects on updates. This pattern provides:

- **Predictable state transitions**
- **Easier testing and debugging**
- **Better alignment with functional programming**
- **Safer design in concurrent or async environments**

Chapter 13.

Event-Driven OOP with Classes

1. Understanding Event Emitters
2. Creating Custom Event-Handling Classes
3. Practical: Building a Simple Event Bus

13 Event-Driven OOP with Classes

13.1 Understanding Event Emitters

Modern JavaScript applications often rely on **event-driven programming** to respond to user interactions, network responses, and other asynchronous events. Rather than executing code in a strict top-down order, event-driven systems react to events as they occur, allowing for more dynamic and interactive behavior.

13.1.1 What Are Event Emitters?

At the heart of event-driven programming in JavaScript is the **Event Emitter** — an object that facilitates communication between different parts of a system through the **observer pattern**.

The **observer pattern** involves two key roles:

- **Subject (Event Emitter):** The object that maintains a list of observers and notifies them when events occur.
- **Observers (Listeners):** Functions or components that respond to specific events when notified.

This pattern promotes **loose coupling**: components don't need to know about each other directly, they just listen for or emit events.

13.1.2 Basic Event Emitter Methods

A typical event emitter supports the following methods:

Method	Description
`on(event, handler)`	Registers a listener (handler) for a specific event.
`emit(event, data)`	Triggers an event, calling all listeners attached to it.
`off(event, handler)`	Removes a specific listener from an event.

13.1.3 Example: Simple Event Emitter

Let's implement a minimal custom event emitter:

```
class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(event, handler) {
    if (!this.events[event]) {
      this.events[event] = [];
    }
    this.events[event].push(handler);
  }

  emit(event, data) {
    if (this.events[event]) {
      this.events[event].forEach(handler => handler(data));
    }
  }

  off(event, handler) {
    if (this.events[event]) {
      this.events[event] = this.events[event].filter(h => h !== handler);
    }
  }
}


// Using the EventEmitter

const emitter = new EventEmitter();

function onLogin(user) {
  console.log(`User logged in: ${user}`);
}

emitter.on('login', onLogin);
emitter.emit('login', 'alice'); // Output: User logged in: alice

emitter.off('login', onLogin);
emitter.emit('login', 'bob'); // No output, listener removed
```

13.1.4 Benefits of Using Event Emitters

- **Decoupling:** Components don't need to call each other directly. They can simply emit and respond to events.
- **Modularity:** Logic is split into smaller, manageable parts that react to specific triggers.
- **Reusability:** Event emitters can be mixed into any class to add event capabilities.
- **Flexibility:** You can add, remove, or change event handlers without altering core logic.

13.1.5 Common Use Cases

- UI components listening for user interactions (e.g., `click`, `submit`)
- Communication between services in a frontend app
- Plugin systems and hooks
- Logging and analytics tracking

13.1.6 Summary

Event emitters bring the power of the observer pattern to JavaScript, enabling parts of your codebase to **communicate without tight dependencies**. By supporting methods like `on`, `emit`, and `off`, they provide a clean and flexible way to build **modular, maintainable**, and **scalable** systems. In the next section, you'll learn how to build custom event-handling classes using this pattern in your own applications.

13.2 Creating Custom Event-Handling Classes

In object-oriented programming, designing **custom event-handling classes** enables components to communicate through events in a modular and flexible way. These classes can register listeners, emit events, and remove handlers—all essential for building dynamic applications that respond to user input, internal changes, or asynchronous events.

13.2.1 Why Build Custom Event Classes?

While JavaScript environments like Node.js offer built-in event systems (e.g., `EventEmitter`), in many frontend or cross-platform projects you may need to create your own event system. Doing so allows you to:

- Keep event logic encapsulated inside classes.
- Enable any class to emit or react to events.
- Improve modularity and separation of concerns.

13.2.2 Event System Design

A robust event-handling class should support:

- **Registration (`on`)**: Attach a handler for a named event.

- **Emission (`emit`)**: Trigger an event with optional data.
- **Deregistration (`off`)**: Remove a specific handler from a named event.
- **One-time listeners (`once`)** *(optional)*: Attach a listener that runs only once.

13.2.3 Implementing a Custom Event Class

Let's walk through the creation of a reusable `EventEmitter` class.

```
class EventEmitter {
  constructor() {
    this.events = new Map(); // Map to store event types and their listeners
  }

  on(event, listener) {
    if (!this.events.has(event)) {
      this.events.set(event, []);
    }
    this.events.get(event).push(listener);
  }

  emit(event, data) {
    const listeners = this.events.get(event);
    if (listeners) {
      listeners.forEach(listener => listener(data));
    }
  }

  off(event, listener) {
    const listeners = this.events.get(event);
    if (listeners) {
      this.events.set(event, listeners.filter(l => l !== listener));
    }
  }

  once(event, listener) {
    const wrapper = (data) => {
      listener(data);
      this.off(event, wrapper);
    };
    this.on(event, wrapper);
  }
}
```

13.2.4 Example Usage in a Custom Class

You can extend or embed the `EventEmitter` in any custom class:

```
class ChatRoom extends EventEmitter {
  constructor(name) {
    super();
    this.name = name;
  }

  message(user, text) {
    const msg = `[${this.name}] ${user}: ${text}`;
    this.emit('message', msg);
  }
}

const room = new ChatRoom('general');

function logMessage(msg) {
  console.log('Received:', msg);
}

room.on('message', logMessage);
room.message('Alice', 'Hello!'); // Received: [general] Alice: Hello!
room.off('message', logMessage);
room.message('Bob', 'Still there?'); // No output
```

Full runnable code:

```
class EventEmitter {
  constructor() {
    this.events = new Map(); // Map to store event types and their listeners
  }

  on(event, listener) {
    if (!this.events.has(event)) {
      this.events.set(event, []);
    }
    this.events.get(event).push(listener);
  }

  emit(event, data) {
    const listeners = this.events.get(event);
    if (listeners) {
      listeners.forEach(listener => listener(data));
    }
  }

  off(event, listener) {
    const listeners = this.events.get(event);
    if (listeners) {
      this.events.set(event, listeners.filter(l => l !== listener));
    }
  }

  once(event, listener) {
    const wrapper = (data) => {
      listener(data);
      this.off(event, wrapper);
    };
    this.on(event, wrapper);
```

```
  }
}

class ChatRoom extends EventEmitter {
  constructor(name) {
    super();
    this.name = name;
  }

  message(user, text) {
    const msg = `[${this.name}] ${user}: ${text}`;
    this.emit('message', msg);
  }
}

const room = new ChatRoom('general');

function logMessage(msg) {
  console.log('Received:', msg);
}

room.on('message', logMessage);
room.message('Alice', 'Hello!'); // Received: [general] Alice: Hello!
room.off('message', logMessage);
room.message('Bob', 'Still there?'); // No output
```

13.2.5 Managing Multiple Events and Memory

Using a `Map` instead of an object for storing listeners helps prevent name collisions. To manage memory effectively:

- **Remove listeners** when they're no longer needed using `off()`.
- Use `once()` for handlers that should run only once.
- Periodically audit your application to ensure listeners aren't persisting longer than necessary.

Memory Leak Tip: Long-lived listeners on short-lived objects (like DOM elements in SPA frameworks) can lead to memory leaks if not explicitly removed.

13.2.6 Best Practices

- **Avoid duplicate listeners**: Optionally check for existing handlers before adding.
- **Encapsulate event logic**: Don't expose internal details through events unless necessary.
- **Prefer weak references** *(in advanced cases)* to allow garbage collection (not shown here for simplicity).

13.2.7 Summary

Custom event-handling classes enable clean, loosely coupled communication within your JavaScript applications. By implementing common event operations like `on`, `emit`, `off`, and `once`, your classes can broadcast changes, coordinate behavior, and enhance interactivity—all while maintaining clarity and modularity. In the next section, we'll apply these principles to build a simple yet powerful **event bus**.

13.3 Practical: Building a Simple Event Bus

In this section, you'll learn how to build a **simple event bus**—a central messaging hub that facilitates communication between different components of your application. This is especially useful in decoupled architectures, where components should not directly depend on each other but still need to react to events.

13.3.1 What Is an Event Bus?

An **event bus** is a shared object or module that enables objects to:

- **Emit events** when something happens.
- **Listen to events** to respond to changes or actions elsewhere.

It implements the **observer pattern**, where multiple listeners observe and respond to changes from a shared publisher (the bus).

13.3.2 Building a Simple EventBus Class

We'll build an `EventBus` class from scratch. It will support:

- `on(event, listener)`: Register a listener for an event.
- `off(event, listener)`: Remove a specific listener.
- `emit(event, data)`: Notify all listeners of an event.
- `once(event, listener)`: Register a listener that fires only once.

```
class EventBus {
  constructor() {
    this.listeners = new Map(); // Stores event => [listeners]
  }

  on(event, handler) {
    if (!this.listeners.has(event)) {
```

```
      this.listeners.set(event, []);
    }
    this.listeners.get(event).push(handler);
  }

  emit(event, payload) {
    const handlers = this.listeners.get(event);
    if (handlers) {
      handlers.forEach(handler => handler(payload));
    }
  }

  off(event, handler) {
    const handlers = this.listeners.get(event);
    if (handlers) {
      const filtered = handlers.filter(h => h !== handler);
      this.listeners.set(event, filtered);
    }
  }

  once(event, handler) {
    const wrapper = (payload) => {
      handler(payload);
      this.off(event, wrapper);
    };
    this.on(event, wrapper);
  }
}
```

13.3.3 Example: Communicating Between Components

Let's simulate two components—**UserForm** and **UserList**—communicating via the event bus:

```
const bus = new EventBus();

// UserList component
class UserList {
  constructor() {
    this.users = [];
    bus.on('user:added', this.addUser.bind(this));
  }

  addUser(user) {
    this.users.push(user);
    console.log(`User added: ${user.name}`);
    this.display();
  }

  display() {
    console.log("Current Users:", this.users.map(u => u.name).join(', '));
  }
}
```

```
// UserForm component
class UserForm {
  submit(name) {
    const user = { name };
    console.log(`Submitting user: ${name}`);
    bus.emit('user:added', user);
  }
}

// Usage
const userList = new UserList();
const form = new UserForm();

form.submit('Alice');
// Output:
// Submitting user: Alice
// User added: Alice
// Current Users: Alice

form.submit('Bob');
// Output:
// Submitting user: Bob
// User added: Bob
// Current Users: Alice, Bob
```

13.3.4 Avoiding Memory Leaks

When components are no longer in use, be sure to call `off()` to remove listeners:

```
// Later, if UserList is destroyed:
bus.off('user:added', userList.addUser.bind(userList));
// Won't work as intended due to new binding
```

Tip: Store listener references before registering so they can be removed properly.

Full runnable code:

```
class EventBus {
  constructor() {
    this.listeners = new Map(); // Stores event => [listeners]
  }

  on(event, handler) {
    if (!this.listeners.has(event)) {
      this.listeners.set(event, []);
    }
    this.listeners.get(event).push(handler);
  }

  emit(event, payload) {
    const handlers = this.listeners.get(event);
    if (handlers) {
```

```
      handlers.forEach(handler => handler(payload));
    }
  }

  off(event, handler) {
    const handlers = this.listeners.get(event);
    if (handlers) {
      const filtered = handlers.filter(h => h !== handler);
      this.listeners.set(event, filtered);
    }
  }

  once(event, handler) {
    const wrapper = (payload) => {
      handler(payload);
      this.off(event, wrapper);
    };
    this.on(event, wrapper);
  }
}
const bus = new EventBus();

// UserList component
class UserList {
  constructor() {
    this.users = [];
    bus.on('user:added', this.addUser.bind(this));
  }

  addUser(user) {
    this.users.push(user);
    console.log(`User added: ${user.name}`);
    this.display();
  }

  display() {
    console.log("Current Users:", this.users.map(u => u.name).join(', '));
  }
}

// UserForm component
class UserForm {
  submit(name) {
    const user = { name };
    console.log(`Submitting user: ${name}`);
    bus.emit('user:added', user);
  }
}

// Usage
const userList = new UserList();
const form = new UserForm();

form.submit('Alice');
// Output:
// Submitting user: Alice
// User added: Alice
// Current Users: Alice
```

```
form.submit('Bob');
// Output:
// Submitting user: Bob
// User added: Bob
// Current Users: Alice, Bob
```

13.3.5 Summary

An **event bus** promotes **loose coupling** by allowing components to communicate without direct references. This technique is particularly useful in:

- Modular frontend frameworks (e.g., Vue, React)
- Pub-sub architectures
- Game loops and GUI state handling

By implementing your own `EventBus` class, you gain fine-grained control over your application's event system and deepen your understanding of event-driven object-oriented programming.

In larger applications, consider extending this idea with namespacing, wildcard events, or async event support.

Chapter 14.

Working with DOM Using OOP

1. Encapsulating DOM Manipulation in Classes
2. Event Handling with Class Methods
3. Practical: Building a To-Do List Application

14 Working with DOM Using OOP

14.1 Encapsulating DOM Manipulation in Classes

In traditional JavaScript, working directly with the DOM often leads to tightly coupled, difficult-to-maintain code. By applying object-oriented principles to DOM manipulation, we can create **modular, reusable components** that are easier to manage, extend, and test.

This section demonstrates how to **encapsulate DOM behavior** inside classes, turning UI elements into objects with clear responsibilities and internal state management.

14.1.1 Why Encapsulate DOM Logic?

Encapsulation allows you to:

- **Bundle state and behavior** related to a specific UI element in one place.
- **Avoid polluting global scope** with functions and variables.
- **Reuse code** by creating multiple instances of a component.
- **Improve readability and maintainability**.

Instead of scattered query selectors and event handlers, class-based DOM components centralize logic.

14.1.2 Structure of a DOM-Encapsulating Class

A typical DOM component class includes:

- A constructor that accepts or queries a DOM element.
- Private fields or properties for managing internal state.
- Methods to update, render, or manipulate the element.
- Optionally, methods for attaching or detaching event listeners.

14.1.3 Example: ToggleButton Component

```
<button id="myToggle">Click Me</button>
```

```
class ToggleButton {
  constructor(selector) {
    this.button = document.querySelector(selector);
```

```
    this.active = false;

    this.button.addEventListener('click', () => this.toggle());
    this.render();
  }

  toggle() {
    this.active = !this.active;
    this.render();
  }

  render() {
    this.button.textContent = this.active ? 'Active' : 'Inactive';
    this.button.classList.toggle('active', this.active);
  }
}

// Usage
const toggle = new ToggleButton('#myToggle');
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Toggle Button Example</title>
  <style>
    button {
      padding: 10px 20px;
      font-size: 16px;
    }

    .active {
      background-color: green;
      color: white;
    }
  </style>
</head>
<body>

  <!-- Step 1: Button Element -->
  <button id="myToggle">Click Me</button>

  <!-- Step 2: JavaScript Logic -->
  <script>
    class ToggleButton {
      constructor(selector) {
        this.button = document.querySelector(selector);
        this.active = false;

        this.button.addEventListener('click', () => this.toggle());
        this.render();
      }

      toggle() {
        this.active = !this.active;
        this.render();
      }
```

```
      render() {
        this.button.textContent = this.active ? 'Active' : 'Inactive';
        this.button.classList.toggle('active', this.active);
      }
    }

    // Usage
    const toggle = new ToggleButton('#myToggle');
  </script>

</body>
</html>
```

Key Concepts Illustrated:

- The DOM query is isolated in the constructor.
- State (`this.active`) is stored on the instance.
- `render()` ensures the DOM reflects the current state.
- Logic is contained in class methods, making it modular.

14.1.4 Techniques for Encapsulation

Here are common tasks and how to encapsulate them:

Querying Elements Within a Component

Use scoped queries to prevent selecting unrelated DOM nodes:

```
this.input = this.root.querySelector('input[type="text"]');
```

Updating Text or Attributes

Encapsulate changes with clearly named methods:

```
setText(message) {
  this.label.textContent = message;
}

setDisabled(state) {
  this.button.disabled = state;
}
```

Controlling Visibility or Styles

```
show() {
  this.root.style.display = 'block';
}
```

```
hide() {
  this.root.style.display = 'none';
}
```

Managing Events

You can add/remove listeners inside the class:

```
attachEvents() {
  this.button.addEventListener('click', this.onClick);
}

detachEvents() {
  this.button.removeEventListener('click', this.onClick);
}
```

14.1.5 Component Composition

Classes can contain other classes or manage child components:

```
class Form {
  constructor(selector) {
    this.root = document.querySelector(selector);
    this.toggleBtn = new ToggleButton('#formToggle');
  }
}
```

This allows building **hierarchical UI components**, a pattern used in frameworks like React, Vue, and Angular.

14.1.6 Summary

Encapsulating DOM manipulation within classes brings structure and maintainability to client-side code. It aligns with object-oriented programming principles by:

- Representing UI elements as objects.
- Managing behavior and state inside methods.
- Supporting reuse and extension via inheritance or composition.

In the next sections, you'll learn how to handle events within class methods and then apply all this knowledge in a practical to-do list application.

14.2 Event Handling with Class Methods

In an object-oriented approach to DOM programming, classes often need to respond to user interactions such as clicks, key presses, or form submissions. This section shows how to **attach event listeners within class instances** and manage them properly, with a focus on handling `this` context correctly and avoiding memory leaks.

14.2.1 Why Handle Events Inside Classes?

Encapsulating event handling within classes:

- Keeps logic close to the elements it controls.
- Makes the component self-contained and reusable.
- Allows proper management of event listener lifecycle (attach/remove).

14.2.2 Understanding `this` in Class Methods

JavaScript's `this` keyword can be tricky inside event handlers. In class methods, `this` refers to the class instance **only if properly bound**. Without binding, `this` inside a listener may point to the DOM element or become `undefined` in strict mode.

14.2.3 Approach 1: Binding in the Constructor

Use `Function.prototype.bind()` to explicitly bind `this`:

```
class ClickCounter {
  constructor(selector) {
    this.button = document.querySelector(selector);
    this.count = 0;

    // Bind event handler
    this.handleClick = this.handleClick.bind(this);
    this.button.addEventListener('click', this.handleClick);
  }

  handleClick() {
    this.count++;
    console.log(`Clicked ${this.count} times`);
  }

  destroy() {
    // Clean up
    this.button.removeEventListener('click', this.handleClick);
```

```
  }
}
```

Pros:

- Explicit and compatible with older environments.
- Works with `removeEventListener`.

14.2.4 Approach 2: Arrow Functions as Methods

Arrow functions capture the lexical `this`, so they preserve context automatically:

```
class HoverBox {
  constructor(selector) {
    this.box = document.querySelector(selector);
    this.box.addEventListener('mouseenter', this.onHover);
  }

  onHover = () => {
    this.box.classList.add('hovered');
  };

  destroy() {
    this.box.removeEventListener('mouseenter', this.onHover);
  }
}
```

Pros:

- No need to manually bind.
- Cleaner syntax.

Caveat:

- Each instance gets a unique function — not shared on prototype.

14.2.5 Common Mistake: Unbound Methods

```
this.button.addEventListener('click', this.handleClick); // NO Not bound
```

This will cause `this` inside `handleClick()` to point to the button element instead of the class instance.

14.2.6 Best Practices for Listener Management

1. **Always bind or use arrow functions** to preserve `this`.
2. **Remove listeners** when the component is no longer used.
3. **Use named handlers**, so they can be unregistered later.

14.2.7 Example: Toggle Button with Cleanup

```
class ToggleButton {
  constructor(selector) {
    this.button = document.querySelector(selector);
    this.active = false;

    this.handleClick = this.handleClick.bind(this);
    this.button.addEventListener('click', this.handleClick);
  }

  handleClick() {
    this.active = !this.active;
    this.button.textContent = this.active ? 'On' : 'Off';
  }

  destroy() {
    // Prevent memory leaks
    this.button.removeEventListener('click', this.handleClick);
  }
}

// Usage
const toggle = new ToggleButton('#myButton');

// Later, when no longer needed
// toggle.destroy();
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Toggle Button with Cleanup</title>
  <style>
    button {
      padding: 10px 20px;
      font-size: 16px;
      margin-right: 10px;
    }
  </style>
</head>
<body>

  <!-- Buttons -->
  <button id="myButton">Off</button>
```

```
  <button id="destroyButton">Destroy Toggle</button>

  <!-- JavaScript -->
  <script>
    class ToggleButton {
      constructor(selector) {
        this.button = document.querySelector(selector);
        this.active = false;

        this.handleClick = this.handleClick.bind(this);
        this.button.addEventListener('click', this.handleClick);
      }

      handleClick() {
        this.active = !this.active;
        this.button.textContent = this.active ? 'On' : 'Off';
      }

      destroy() {
        this.button.removeEventListener('click', this.handleClick);
        this.button.textContent = 'Destroyed';
        this.button.disabled = true;
      }
    }

    // Usage
    const toggle = new ToggleButton('#myButton');

    // Destroy button logic
    document.querySelector('#destroyButton').addEventListener('click', () => {
      toggle.destroy();
    });
  </script>

</body>
</html>
```

14.2.8 Advanced: Delegating Multiple Events

When a component has many listeners or dynamic children, consider using a **single delegated event handler** and filtering by `event.target`.

14.2.9 Summary

Proper event handling inside classes is essential for building maintainable, modular components. Key takeaways:

- Bind methods in the constructor or use arrow functions to ensure the correct `this`.
- Always remove listeners in a `destroy()` or cleanup method.

- Prefer self-contained components that manage their own listeners internally.

Next, we'll apply this knowledge in a real-world project by building a complete To-Do List application using object-oriented DOM patterns.

14.3 Practical: Building a To-Do List Application

In this section, we'll put together everything we've learned about object-oriented programming and DOM manipulation to build a fully functional **To-Do List application**.

We'll use JavaScript **classes** to encapsulate:

- **Task data** and logic
- **UI updates and rendering**
- **Event handling**

The result will be a modular, maintainable, and extensible application.

14.3.1 Features of Our To-Do App

- Add new tasks
- Mark tasks as complete
- Delete tasks
- Clean structure using OOP principles

14.3.2 Application Structure

We'll define two classes:

1. `Task`: Encapsulates a single task's data and DOM.
2. `TodoApp`: Manages the list of tasks and user interaction.

14.3.3 HTML Setup

```
<div id="todo-app">
  <input type="text" id="task-input" placeholder="Enter a new task" />
  <button id="add-task-btn">Add Task</button>
  <ul id="task-list"></ul>
```

```
</div>
```

14.3.4 Class: `Task`

```
class Task {
  constructor(text, onDelete) {
    this.text = text;
    this.completed = false;
    this.onDelete = onDelete;

    this.element = this.createElement();
  }

  createElement() {
    const li = document.createElement('li');
    li.className = 'task';

    this.checkbox = document.createElement('input');
    this.checkbox.type = 'checkbox';
    this.checkbox.addEventListener('change', () => this.toggleComplete());

    this.label = document.createElement('span');
    this.label.textContent = this.text;

    this.deleteBtn = document.createElement('button');
    this.deleteBtn.textContent = 'Delete';
    this.deleteBtn.addEventListener('click', () => this.delete());

    li.append(this.checkbox, this.label, this.deleteBtn);
    return li;
  }

  toggleComplete() {
    this.completed = this.checkbox.checked;
    this.label.style.textDecoration = this.completed ? 'line-through' : 'none';
  }

  delete() {
    this.element.remove();
    this.onDelete(this);
  }
}
```

14.3.5 Class: `TodoApp`

```
class TodoApp {
  constructor(containerId) {
```

```
    this.container = document.querySelector(containerId);
    this.taskInput = this.container.querySelector('#task-input');
    this.addTaskBtn = this.container.querySelector('#add-task-btn');
    this.taskList = this.container.querySelector('#task-list');

    this.tasks = [];

    this.addTaskBtn.addEventListener('click', () => this.addTask());
    this.taskInput.addEventListener('keydown', (e) => {
      if (e.key === 'Enter') this.addTask();
    });
  }

  addTask() {
    const text = this.taskInput.value.trim();
    if (!text) return;

    const task = new Task(text, (taskToDelete) => this.removeTask(taskToDelete));
    this.tasks.push(task);
    this.taskList.appendChild(task.element);
    this.taskInput.value = '';
  }

  removeTask(taskToDelete) {
    this.tasks = this.tasks.filter((task) => task !== taskToDelete);
  }
}
```

14.3.6 Usage

```
document.addEventListener('DOMContentLoaded', () => {
  const app = new TodoApp('#todo-app');
});
```

14.3.7 Optional CSS for Styling

```
.task {
  display: flex;
  align-items: center;
  gap: 10px;
  margin-bottom: 5px;
}

.task button {
  margin-left: auto;
}
```

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

    #todo-app {
      max-width: 400px;
      margin: auto;
    }

    #task-input {
      width: 70%;
      padding: 8px;
      font-size: 16px;
    }

    #add-task-btn {
      padding: 8px 12px;
      font-size: 16px;
    }

    #task-list {
      list-style: none;
      padding: 0;
      margin-top: 20px;
    }

    .task {
      display: flex;
      align-items: center;
      gap: 10px;
      margin-bottom: 5px;
    }

    .task button {
      margin-left: auto;
    }
  </style>
</head>
<body>

  <!-- HTML Setup -->
  <div id="todo-app">
    <input type="text" id="task-input" placeholder="Enter a new task" />
    <button id="add-task-btn">Add Task</button>
    <ul id="task-list"></ul>
  </div>

  <!-- JavaScript Logic -->
  <script>
    // Class: Task
    class Task {
```

```
    constructor(text, onDelete) {
      this.text = text;
      this.completed = false;
      this.onDelete = onDelete;

      this.element = this.createElement();
    }

    createElement() {
      const li = document.createElement('li');
      li.className = 'task';

      this.checkbox = document.createElement('input');
      this.checkbox.type = 'checkbox';
      this.checkbox.addEventListener('change', () => this.toggleComplete());

      this.label = document.createElement('span');
      this.label.textContent = this.text;

      this.deleteBtn = document.createElement('button');
      this.deleteBtn.textContent = 'Delete';
      this.deleteBtn.addEventListener('click', () => this.delete());

      li.append(this.checkbox, this.label, this.deleteBtn);
      return li;
    }

    toggleComplete() {
      this.completed = this.checkbox.checked;
      this.label.style.textDecoration = this.completed ? 'line-through' : 'none';
    }

    delete() {
      this.element.remove();
      this.onDelete(this);
    }
  }

  // Class: TodoApp
  class TodoApp {
    constructor(containerId) {
      this.container = document.querySelector(containerId);
      this.taskInput = this.container.querySelector('#task-input');
      this.addTaskBtn = this.container.querySelector('#add-task-btn');
      this.taskList = this.container.querySelector('#task-list');

      this.tasks = [];

      this.addTaskBtn.addEventListener('click', () => this.addTask());
      this.taskInput.addEventListener('keydown', (e) => {
        if (e.key === 'Enter') this.addTask();
      });
    }

    addTask() {
      const text = this.taskInput.value.trim();
      if (!text) return;
```

```
        const task = new Task(text, (taskToDelete) => this.removeTask(taskToDelete));
        this.tasks.push(task);
        this.taskList.appendChild(task.element);
        this.taskInput.value = '';
      }

      removeTask(taskToDelete) {
        this.tasks = this.tasks.filter((task) => task !== taskToDelete);
      }
    }

    // Usage
    document.addEventListener('DOMContentLoaded', () => {
      const app = new TodoApp('#todo-app');
    });
  </script>

</body>
</html>
```

14.3.8 Benefits of the OOP Approach

- **Encapsulation**: Each task manages its own DOM and logic.
- **Separation of concerns**: The app logic is cleanly separated from UI rendering.
- **Reusability**: The `Task` class can be reused in other task-based apps.
- **Maintainability**: Easy to extend with features like persistence or editing.

14.3.9 Summary

Using classes to structure a DOM-based app results in modular, reusable, and testable code. This To-Do list example demonstrates how to combine:

- Object state (`Task`)
- DOM control (`createElement`, event listeners)
- Event-driven updates (`TodoApp`)

In the next chapters, we'll explore more advanced topics like meta-programming and decorators, building on this solid OOP foundation.

Chapter 15.

Asynchronous Programming and OOP

1. Promises and Async/Await Overview
2. Using OOP to Manage Asynchronous Code
3. Practical: Async Data Fetching Class

15 Asynchronous Programming and OOP

15.1 Promises and Async/Await Overview

15.1.1 What is Asynchronous Programming?

In JavaScript, **asynchronous programming** allows your code to perform long-running operations—like fetching data from a server or reading files—without blocking the rest of the program. This means your application remains responsive while waiting for these operations to complete.

15.1.2 The Problem with Callbacks

Historically, JavaScript handled async operations using **callbacks**—functions passed as arguments that run when an operation finishes. However, callbacks often lead to complex and hard-to-read code, especially when you have multiple async steps depending on each other, a problem sometimes called *"callback hell"*.

15.1.3 Enter Promises

A **Promise** is a JavaScript object representing the eventual completion (or failure) of an asynchronous operation and its resulting value.

Key Points:

- A Promise can be in one of three states:
  - **Pending**: The operation is ongoing.
  - **Fulfilled**: The operation completed successfully.
  - **Rejected**: The operation failed.
- Promises provide a clean way to chain async operations and handle errors.

15.1.4 Creating and Using a Promise

```
const myPromise = new Promise((resolve, reject) => {
  // Simulate an async operation (e.g., fetching data)
  setTimeout(() => {
    const success = true;
    if (success) {
```

```
      resolve('Operation successful!');
    } else {
      reject('Operation failed.');
    }
  }, 1000);
});

// Consuming the promise
myPromise
  .then(result => {
    console.log(result); // "Operation successful!"
  })
  .catch(error => {
    console.error(error);
  });
```

15.1.5 Promise Chaining

Promises let you chain multiple async operations in a readable way:

```
fetchData()
  .then(data => processData(data))
  .then(processed => saveData(processed))
  .catch(error => console.error('Error:', error));
```

Each `.then()` returns a new promise, enabling sequential execution.

15.1.6 Async/Await: Syntactic Sugar for Promises

`async/await` is a modern syntax built on Promises that makes asynchronous code look and behave like synchronous code, improving readability.

Example:

```
async function fetchAndProcess() {
  try {
    const data = await fetchData();
    const processed = await processData(data);
    await saveData(processed);
    console.log('All done!');
  } catch (error) {
    console.error('Error:', error);
  }
}
```

- `await` pauses execution until the Promise settles.

- `async` functions always return a Promise.
- Errors can be caught easily with `try...catch`.

15.1.7 Why Use Promises and Async/Await?

- **Avoid callback hell:** Cleaner, more maintainable code.
- **Better error handling:** Centralized with `.catch()` or `try...catch`.
- **Sequential and parallel operations:** Control over async flow.
- **Integration with modern APIs:** Many built-in APIs now return Promises.

15.1.8 Summary

Promises and `async/await` are powerful tools in JavaScript that help manage asynchronous operations cleanly and efficiently. Understanding these concepts is essential for building modern applications that interact with APIs, databases, or perform any delayed computations.

In the next section, we'll explore how to design classes that manage asynchronous tasks, leveraging Promises and async/await to create robust and readable object-oriented code.

15.2 Using OOP to Manage Asynchronous Code

Asynchronous programming is essential in modern JavaScript applications, and combining it effectively with object-oriented design improves code organization, reuse, and maintainability.

When your classes need to perform async tasks—such as fetching data, reading files, or waiting for timers—you can design them to **encapsulate async behavior** while managing state and providing a clear API.

15.2.1 Structuring Classes for Async Tasks

Here are key strategies for managing async operations within classes:

Define Async Methods

Use `async` methods inside classes to perform asynchronous operations, returning Promises that callers can `await` or `.then()`.

```
class DataFetcher {
  async fetchData(url) {
    const response = await fetch(url);
    if (!response.ok) throw new Error('Network response was not ok');
    const data = await response.json();
    return data;
  }
}
```

Maintain Internal State

Classes can store the state related to async operations, like loading status or cached data, enabling consumers to query or react accordingly.

```
class UserProfile {
  constructor() {
    this.data = null;
    this.loading = false;
  }

  async load(userId) {
    this.loading = true;
    try {
      const response = await fetch(`/api/users/${userId}`);
      this.data = await response.json();
    } finally {
      this.loading = false;
    }
  }
}
```

Expose Promises for Async Control

By returning Promises, async methods allow callers to chain actions or handle errors with familiar patterns.

15.2.2 Handling Common Challenges

Concurrency Management

When multiple async operations may run concurrently, classes can implement mechanisms such as:

- **Queues:** To serialize tasks and prevent race conditions.
- **Cancellation tokens or flags:** To abort or ignore outdated async calls.

```
class TaskQueue {
  constructor() {
    this.queue = Promise.resolve();
  }
```

```
  enqueue(task) {
    this.queue = this.queue.then(() => task());
    return this.queue;
  }
}
```

Error Propagation

Async methods should propagate errors naturally by throwing exceptions or rejecting promises. Use `try...catch` internally to handle or log errors without swallowing them silently.

15.2.3 Testing Async Methods

Testing classes with async methods requires handling asynchronous behavior:

- Use test frameworks supporting async/await.
- Await the async method calls to ensure the test waits for completion.
- Mock or stub external async dependencies to isolate tests.

```
test('fetchData returns correct data', async () => {
  const fetcher = new DataFetcher();
  const data = await fetcher.fetchData('https://api.example.com/data');
  expect(data).toHaveProperty('id');
});
```

15.2.4 Summary

Incorporating asynchronous operations in object-oriented design means designing classes that:

- Have **async methods** returning Promises.
- Maintain **internal state** to reflect ongoing or completed operations.
- Handle **errors and concurrency** explicitly.
- Provide clean APIs for consumers to work with asynchronous results.

This approach results in modular, readable, and testable code that leverages JavaScript's async features within an OOP framework.

Next, we'll put these concepts into practice by building an asynchronous data-fetching class with robust state management and error handling.

15.3 Practical: Async Data Fetching Class

In this section, we will build a practical `AsyncDataFetcher` class that demonstrates how to integrate asynchronous data fetching with object-oriented principles. This class will:

- Fetch data asynchronously from an API using `fetch` and `async/await`.
- Manage loading and error states internally.
- Provide methods to access and refresh the fetched data.
- Handle errors gracefully.

15.3.1 Step 1: Defining the Class Structure

Our `AsyncDataFetcher` will have these core properties:

- `data` — stores the fetched data.
- `loading` — indicates if a fetch operation is in progress.
- `error` — stores any error encountered during fetching.

And methods:

- `fetch(url)` — async method to fetch data from the provided URL.
- `getData()` — returns the current data.
- `hasError()` — returns whether an error occurred.
- `isLoading()` — returns loading status.

15.3.2 Step 2: Implementing the Class

```
class AsyncDataFetcher {
  constructor() {
    this.data = null;
    this.loading = false;
    this.error = null;
  }

  // Async method to fetch data from the API
  async fetch(url) {
    this.loading = true;
    this.error = null;
    this.data = null;

    try {
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error(`HTTP error! Status: ${response.status}`);
      }
```

```
      this.data = await response.json();
    } catch (err) {
      this.error = err;
    } finally {
      this.loading = false;
    }
  }

  // Getter for the fetched data
  getData() {
    return this.data;
  }

  // Check if currently loading
  isLoading() {
    return this.loading;
  }

  // Check if an error occurred
  hasError() {
    return this.error !== null;
  }

  // Get error message if any
  getError() {
    return this.error ? this.error.message : null;
  }
}
```

15.3.3 Step 3: Using the Class

Here's how you can create an instance and use it to fetch data asynchronously:

```
async function runExample() {
  const fetcher = new AsyncDataFetcher();

  console.log('Starting fetch...');
  await fetcher.fetch('https://jsonplaceholder.typicode.com/posts/1');

  if (fetcher.hasError()) {
    console.error('Error fetching data:', fetcher.getError());
  } else {
    console.log('Fetched data:', fetcher.getData());
  }

  console.log('Loading status:', fetcher.isLoading());
}

runExample();
```

Full runnable code:

```javascript
// Step 1 & 2: Define and Implement the AsyncDataFetcher Class
class AsyncDataFetcher {
  constructor() {
    this.data = null;
    this.loading = false;
    this.error = null;
  }

  async fetch(url) {
    this.loading = true;
    this.error = null;
    this.data = null;

    try {
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error(`HTTP error! Status: ${response.status}`);
      }

      this.data = await response.json();
    } catch (err) {
      this.error = err;
    } finally {
      this.loading = false;
    }
  }

  getData() {
    return this.data;
  }

  isLoading() {
    return this.loading;
  }

  hasError() {
    return this.error !== null;
  }

  getError() {
    return this.error ? this.error.message : null;
  }
}

// Step 3: Use the Class
async function runExample() {
  const fetcher = new AsyncDataFetcher();

  console.log('Starting fetch...');
  await fetcher.fetch('https://jsonplaceholder.typicode.com/posts/1');

  if (fetcher.hasError()) {
    console.error('Error fetching data:', fetcher.getError());
  } else {
    console.log('Fetched data:', fetcher.getData());
  }
```

```
  console.log('Loading status:', fetcher.isLoading());
}

runExample();
```

15.3.4 How This Works

- The `fetch` method sets `loading` to `true` before starting the asynchronous operation.
- It attempts to fetch JSON data from the given URL.
- On success, it stores the data and resets error state.
- On failure, it captures the error.
- Regardless of outcome, it sets `loading` to `false` at the end.
- The state getters allow external code to inspect current loading, error, or data states safely.

15.3.5 Benefits of This Approach

- **Encapsulation:** The class hides async details and manages state internally.
- **Clear API:** Consumers call `fetch()` and check status via `isLoading()` or `hasError()`.
- **Error Handling:** Centralized error catching inside the class reduces duplication.
- **Reusable:** You can extend or reuse this class for any kind of async data fetching.

15.3.6 Summary

This example highlights how asynchronous programming patterns blend with object-oriented design to produce clean, maintainable, and robust code. Using async/await inside classes simplifies handling complex async flows and keeps state management coherent.

In the next chapter, we'll explore more advanced asynchronous patterns like concurrency control and cancellation within OOP contexts.

Chapter 16.

Design Patterns in JavaScript OOP

1. Singleton Pattern
2. Factory Pattern
3. Observer Pattern
4. Decorator Pattern
5. Practical: Applying Patterns in a Chat Application

16 Design Patterns in JavaScript OOP

16.1 Singleton Pattern

The **Singleton** pattern is a design principle that ensures a class has only **one single instance** throughout the entire application. Moreover, it provides a **global access point** to that instance, allowing consistent and centralized management of shared resources or state.

Think of a Singleton like a **single control tower** at an airport — no matter how many flights there are, all communication passes through one central entity to keep everything coordinated.

16.1.1 Why Use the Singleton Pattern?

Singletons are especially useful when you need to:

- **Manage shared resources** like database connections, logging services, or caches, ensuring there aren't multiple conflicting instances.
- Maintain **application-wide configuration** or state that must be consistent and accessible from various parts of your app.
- Provide a **centralized point of control** without resorting to global variables, which can pollute the global namespace.

16.1.2 Implementing Singletons in JavaScript

JavaScript's module system and closures naturally support the Singleton pattern because:

- Modules are **evaluated once** and cached, so exporting a single instance inherently acts like a singleton.
- Closures allow private state and control over instance creation.

Here is a simple Singleton implemented using a class and an immediately-invoked function expression (IIFE):

```
const Logger = (function () {
  let instance;

  function createInstance() {
    return {
      log: (msg) => console.log(`[LOG]: ${msg}`),
      error: (msg) => console.error(`[ERROR]: ${msg}`),
    };
  }

  return {
```

```
    getInstance() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

// Usage:
const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();

console.log(logger1 === logger2); // true, both are the same instance

logger1.log("Singleton pattern in action!");
\vspace{1em}
```

In this example:

- The `Logger` module keeps a private `instance` variable.
- The first call to `getInstance()` creates and returns the single instance.
- Subsequent calls return the same instance, guaranteeing only one logger exists.

16.1.3 Singleton with ES6 Modules

Thanks to the ES6 module system, you can create singletons simply by exporting an instance directly:

```
// config.js
class Config {
  constructor() {
    this.settings = {};
  }
  set(key, value) {
    this.settings[key] = value;
  }
  get(key) {
    return this.settings[key];
  }
}

const config = new Config();
export default config;
\vspace{1em}
```

Now, importing `config` anywhere in your app refers to the **same instance**, naturally enforcing singleton behavior.

16.1.4 Cautions When Using Singletons

While singletons can be useful, overusing them or using them inappropriately can cause problems:

- **Hidden dependencies:** Global singletons may introduce implicit dependencies, making code harder to understand and maintain.
- **Testing challenges:** Because singletons maintain shared state, tests may interfere with each other unless carefully isolated or reset.
- **Reduced flexibility:** Relying heavily on a single instance can limit your app's ability to scale or support multiple configurations.

Use singletons **sparingly** and document their roles clearly to avoid pitfalls.

16.1.5 Summary

The Singleton pattern is a foundational design approach in JavaScript OOP that restricts instantiation to a single object and provides a global access point. Leveraging closures or modules makes it straightforward to implement. When applied judiciously, singletons simplify resource management and state sharing, but they should be used carefully to maintain clean, testable code.

In the next section, we will explore the **Factory Pattern**, another versatile way to create objects flexibly.

16.2 Factory Pattern

16.2.1 What is the Factory Pattern?

The **Factory pattern** is a creational design pattern that **abstracts the process of object creation**, allowing you to produce instances of different classes based on input parameters or logic, without exposing the creation details to the client code.

Rather than directly instantiating objects with `new`, you use a **factory function or class** to centralize and manage object creation. This approach promotes **loose coupling** and makes your codebase easier to extend and maintain.

Think of a factory like a **car factory**: depending on the model selected, it produces the appropriate car without the buyer needing to know the assembly details.

16.2.2 Why Use the Factory Pattern?

- **Encapsulation of instantiation logic:** You keep the `new` operator and construction logic in one place, not scattered throughout your code.
- **Simplifies client code:** Consumers simply ask the factory for an object of a certain type without worrying about class details.
- **Supports scalability:** Adding new product types only requires updating the factory, not all the client code that creates objects.
- **Improves code maintainability:** Changes to construction logic happen in one centralized location.

16.2.3 Factory Pattern Implementation in JavaScript

Factories can be implemented as simple functions or classes that return new instances based on input parameters.

Example: A Shape Factory

```
// Base classes
class Circle {
  draw() {
    console.log("Drawing a Circle");
  }
}

class Square {
  draw() {
    console.log("Drawing a Square");
  }
}

// Factory function
function shapeFactory(type) {
  switch (type.toLowerCase()) {
    case 'circle':
      return new Circle();
    case 'square':
      return new Square();
    default:
      throw new Error('Unknown shape type');
  }
}

// Usage:
const shapes = [
  shapeFactory('circle'),
  shapeFactory('square'),
  shapeFactory('circle'),
];

shapes.forEach(shape => shape.draw());
```

```
// Output:
// Drawing a Circle
// Drawing a Square
// Drawing a Circle
\vspace{1em}
```

In this example:

- The `shapeFactory` function encapsulates the logic deciding which shape to create.
- The client code simply calls `shapeFactory` with the desired type and works with the returned object polymorphically.
- Adding a new shape requires only adding a new class and extending the factory logic.

16.2.4 Using Factory Classes

You can also create a dedicated factory class to manage object creation:

```
class ShapeFactory {
  createShape(type) {
    switch (type.toLowerCase()) {
      case 'circle':
        return new Circle();
      case 'square':
        return new Square();
      default:
        throw new Error('Unknown shape type');
    }
  }
}

// Usage:
const factory = new ShapeFactory();
const shape1 = factory.createShape('circle');
const shape2 = factory.createShape('square');

shape1.draw(); // Drawing a Circle
shape2.draw(); // Drawing a Square
\vspace{1em}
```

This approach is helpful when your factory needs to maintain internal state or more complex creation logic.

16.2.5 Benefits of the Factory Pattern

- **Loose coupling:** Client code depends only on abstract interfaces or base classes, not concrete implementations.
- **Single responsibility:** Creation logic is separated from business logic.
- **Extensibility:** You can introduce new product types without changing existing client

code.
- **Improved testing:** Factories can be mocked or stubbed during tests for easier isolation.

16.2.6 Summary

The Factory pattern is a powerful way to centralize and abstract object creation in JavaScript. By using factory functions or classes, you decouple your code from concrete implementations and simplify the instantiation process. This pattern is especially useful in applications where multiple related objects need to be created dynamically based on runtime conditions.

16.3 Observer Pattern

16.3.1 What is the Observer Pattern?

The **Observer pattern** is a behavioral design pattern used to implement **event-driven architectures**. It allows one object, called the **subject** (or observable), to maintain a list of dependent objects, called **observers**, and notify them automatically of any state changes or events.

In this pattern, observers **subscribe** to the subject to receive updates or notifications. When the subject's state changes, it **broadcasts** those changes to all subscribed observers, allowing them to react accordingly.

16.3.2 Why Use the Observer Pattern?

- **Decouples components:** Subjects and observers are loosely connected — the subject doesn't need to know the internal details of observers, only that they implement a certain interface (e.g., a callback function).
- **Enables dynamic behavior:** Observers can be added or removed at runtime, allowing flexible reactions to events.
- **Promotes scalability:** Many parts of an application can react to a single event without the subject managing those reactions explicitly.

16.3.3 The Observer Pattern in JavaScript

JavaScript naturally lends itself to this pattern through its event-driven nature. The DOM event system, Node.js's `EventEmitter`, and custom event systems are real-world examples of the Observer pattern.

16.3.4 Key Components

- **Subject:** The source of events or state changes. It maintains a list of observers and provides methods to add, remove, and notify them.
- **Observers:** Objects or functions that subscribe to the subject to receive notifications.

16.3.5 Conceptual Example

```
// Subject class
class Subject {
  constructor() {
    this.observers = [];
  }

  // Add observer (subscribe)
  subscribe(observer) {
    this.observers.push(observer);
  }

  // Remove observer (unsubscribe)
  unsubscribe(observer) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }

  // Notify all observers
  notify(data) {
    this.observers.forEach(observer => observer.update(data));
  }
}

// Observer interface (just an object with update method)
class Observer {
  constructor(name) {
    this.name = name;
  }

  update(data) {
    console.log(`${this.name} received data:`, data);
  }
}

// Usage:
```

```
const subject = new Subject();

const observer1 = new Observer('Observer 1');
const observer2 = new Observer('Observer 2');

subject.subscribe(observer1);
subject.subscribe(observer2);

// Trigger notifications
subject.notify('Event A');
// Output:
// Observer 1 received data: Event A
// Observer 2 received data: Event A

subject.unsubscribe(observer1);

subject.notify('Event B');
// Output:
// Observer 2 received data: Event B
\vspace{1em}
```

16.3.6 Relation to JavaScripts Native Event Systems

- The **DOM event system** follows the Observer pattern where event listeners subscribe to DOM elements, waiting for events like clicks or keypresses.
- Node.js's `EventEmitter` class provides methods like `.on()`, `.emit()`, and `.off()` that implement observer subscription, notification, and unsubscription.
- Custom event emitter implementations mirror the same pattern to enable modular and decoupled components.

16.3.7 Benefits of Using the Observer Pattern

- Promotes **loose coupling** by separating event producers and consumers.
- Supports **flexible and dynamic** event handling by allowing observers to subscribe or unsubscribe at any time.
- Encourages **reusable components** that can interact without direct dependencies.

16.3.8 Summary

The Observer pattern is a fundamental design principle for building scalable, event-driven applications. It allows objects to communicate changes efficiently without tight coupling. JavaScript's event systems are practical realizations of this pattern, and understanding it is

key to mastering modular and reactive programming.

Next, we will dive into the **Decorator Pattern**, a powerful technique for enhancing object behavior dynamically.

16.4 Decorator Pattern

16.4.1 What is the Decorator Pattern?

The **Decorator pattern** is a structural design pattern used to **dynamically add new behavior or responsibilities to objects without modifying their original structure**. Instead of changing the object itself, decorators **wrap** the original object, extending or overriding its functionality in a flexible and reusable way.

This pattern allows behavior to be added transparently and composed at runtime, avoiding the pitfalls of subclassing for every possible combination of features.

16.4.2 Why Use the Decorator Pattern?

- **Extends behavior without inheritance:** Instead of creating complex inheritance hierarchies, decorators wrap objects to add new capabilities.
- **Promotes single responsibility:** Each decorator can focus on a specific enhancement, making code modular and maintainable.
- **Enables runtime flexibility:** Behaviors can be added or removed dynamically, based on runtime conditions.
- **Supports cross-cutting concerns:** Common features like logging, validation, or caching can be applied cleanly without cluttering core logic.

16.4.3 How Decorators Work

A decorator typically holds a reference to the original object and implements the same interface or API. When a method is called on the decorator, it can perform additional actions before or after delegating the call to the original object.

16.4.4 Simple Example: Decorating a Method for Logging

```
// Original class
class Calculator {
  add(a, b) {
    return a + b;
  }
}

// Decorator function that adds logging to any method
function logDecorator(originalObj, methodName) {
  const originalMethod = originalObj[methodName];

  originalObj[methodName] = function(...args) {
    console.log(`Calling ${methodName} with arguments:`, args);
    const result = originalMethod.apply(this, args);
    console.log(`Result of ${methodName}:`, result);
    return result;
  };

  return originalObj;
}

// Usage
const calc = new Calculator();
logDecorator(calc, 'add');

calc.add(2, 3);
// Output:
// Calling add with arguments: [2, 3]
// Result of add: 5
\vspace{1em}
```

In this example, the `logDecorator` wraps the `add` method of `Calculator` to inject logging behavior without modifying the class's code.

16.4.5 Decorating Properties and Validations

Decorators can also be used to enhance properties, such as adding validation before setting a value:

```
function validatePositiveDecorator(obj, propertyName) {
  let value = obj[propertyName];

  Object.defineProperty(obj, propertyName, {
    get() {
      return value;
    },
    set(newVal) {
      if (newVal < 0) {
        throw new Error(`${propertyName} must be positive`);
      }
      value = newVal;
```

```
    },
    configurable: true,
    enumerable: true
  });

  return obj;
}

const product = { price: 10 };
validatePositiveDecorator(product, 'price');

product.price = 20;  // Works fine
product.price = -5;  // Throws Error: price must be positive
\vspace{1em}
```

16.4.6 Decorators vs. Subclassing

While subclassing adds behavior statically at design time, decorators add behavior dynamically at runtime and can be composed or stacked to combine multiple enhancements.

16.4.7 Summary

The Decorator pattern is a powerful technique to **augment objects flexibly without modifying their core implementations**. By wrapping objects, decorators enable the addition of cross-cutting concerns like logging, validation, caching, or more, promoting modular, maintainable, and reusable code.

Next, we will apply these design patterns in a practical chat application example.

16.5 Practical: Applying Patterns in a Chat Application

In this section, we'll build a simplified chat application demonstrating how several design patterns can work together seamlessly:

- **Singleton**: Manage global app state
- **Factory**: Create different types of messages (text, image)
- **Observer**: Notify UI components about new messages
- **Decorator**: Enhance messages with additional behavior (e.g., formatting or encryption)

16.5.1 Step 1: Singleton ChatApp State Manager

We want a single shared instance managing the chat state (messages, users, etc.). This prevents conflicting data and ensures centralized access.

```
class ChatApp {
  constructor() {
    if (ChatApp.instance) {
      return ChatApp.instance;
    }
    this.messages = [];
    this.listeners = [];
    ChatApp.instance = this;
  }

  addMessage(message) {
    this.messages.push(message);
    this.notify(message);
  }

  subscribe(listener) {
    this.listeners.push(listener);
  }

  unsubscribe(listener) {
    this.listeners = this.listeners.filter(l => l !== listener);
  }

  notify(message) {
    this.listeners.forEach(listener => listener.update(message));
  }
}

// Usage: always get the same instance
const chatApp = new ChatApp();
const chatApp2 = new ChatApp();
console.log(chatApp === chatApp2);  // true
\vspace{1em}
```

16.5.2 Step 2: Factory Message Creation

We need a flexible way to create different message types without cluttering the client code.

```
// Base Message class
class Message {
  constructor(sender, content) {
    this.sender = sender;
    this.content = content;
    this.timestamp = new Date();
  }

  display() {
    return `[${this.timestamp.toLocaleTimeString()}] ${this.sender}: ${this.content}`;
  }
```

```
}

// TextMessage subclass
class TextMessage extends Message {
  constructor(sender, content) {
    super(sender, content);
  }
}

// ImageMessage subclass
class ImageMessage extends Message {
  constructor(sender, imageUrl) {
    super(sender, `[Image: ${imageUrl}]`);
    this.imageUrl = imageUrl;
  }
}

// Factory function
function MessageFactory(type, sender, content) {
  switch(type) {
    case 'text':
      return new TextMessage(sender, content);
    case 'image':
      return new ImageMessage(sender, content);
    default:
      throw new Error(`Unknown message type: ${type}`);
  }
}
\vspace{1em}
```

16.5.3 Step 3: Observer UI Listener for New Messages

Our UI components can subscribe to new messages and update automatically when notified.

```
class MessageListView {
  update(message) {
    console.log('New message received: ', message.display());
    // In a real app, update DOM here instead of console.log
  }
}

// Subscribe to chat app notifications
const messageListView = new MessageListView();
chatApp.subscribe(messageListView);
\vspace{1em}
```

16.5.4 Step 4: Decorator Enhancing Messages

Decorators let us extend messages dynamically, e.g., add encryption or formatting.

```
// Decorator to add emoji to text messages
function emojiDecorator(message) {
  const originalDisplay = message.display.bind(message);
  message.display = function() {
    return originalDisplay() + '  ';
  };
  return message;
}

// Decorator to simulate encryption by scrambling content
function encryptDecorator(message) {
  const originalContent = message.content;
  message.content = originalContent.split('').reverse().join('');

  const originalDisplay = message.display.bind(message);
  message.display = function() {
    return originalDisplay() + '  ';
  };
  return message;
}
\vspace{1em}
```

16.5.5 Step 5: Putting It All Together

Here's how the patterns integrate into real usage:

```
// Get the singleton app instance
const app = new ChatApp();

// Subscribe UI component
const ui = new MessageListView();
app.subscribe(ui);

// Create messages via factory
let msg1 = MessageFactory('text', 'Alice', 'Hello!');
let msg2 = MessageFactory('image', 'Bob', 'http://image.url/pic.jpg');

// Decorate messages dynamically
msg1 = emojiDecorator(msg1);
msg2 = encryptDecorator(msg2);

// Add messages to app, which notifies observers
app.addMessage(msg1);
app.addMessage(msg2);
\vspace{1em}
```

Console output:

```
New message received:  [12:00:00 PM] Alice: Hello!
New message received:  [12:00:01 PM] Bob: ]gpj.cip/lru.egami//:ptth[
\vspace{1em}
```

```
// chat-app.js

// Step 1: Singleton - ChatApp State Manager
class ChatApp {
  constructor() {
    if (ChatApp.instance) {
      return ChatApp.instance;
    }
    this.messages = [];
    this.listeners = [];
    ChatApp.instance = this;
  }

  addMessage(message) {
    this.messages.push(message);
    this.notify(message);
  }

  subscribe(listener) {
    this.listeners.push(listener);
  }

  unsubscribe(listener) {
    this.listeners = this.listeners.filter(l => l !== listener);
  }

  notify(message) {
    this.listeners.forEach(listener => listener.update(message));
  }
}

// Step 2: Factory - Message Creation
class Message {
  constructor(sender, content) {
    this.sender = sender;
    this.content = content;
    this.timestamp = new Date();
  }

  display() {
    return `[${this.timestamp.toLocaleTimeString()}] ${this.sender}: ${this.content}`;
  }
}

class TextMessage extends Message {
  constructor(sender, content) {
    super(sender, content);
  }
}

class ImageMessage extends Message {
  constructor(sender, imageUrl) {
    super(sender, `[Image: ${imageUrl}]`);
    this.imageUrl = imageUrl;
  }
}

function MessageFactory(type, sender, content) {
```

```
  switch (type) {
    case 'text':
      return new TextMessage(sender, content);
    case 'image':
      return new ImageMessage(sender, content);
    default:
      throw new Error(`Unknown message type: ${type}`);
  }
}

// Step 3: Observer - UI Listener
class MessageListView {
  update(message) {
    console.log('New message received:', message.display());
  }
}

// Step 4: Decorator - Enhancing Messages
function emojiDecorator(message) {
  const originalDisplay = message.display.bind(message);
  message.display = function () {
    return originalDisplay() + '  ';
  };
  return message;
}

function encryptDecorator(message) {
  const originalContent = message.content;
  message.content = originalContent.split('').reverse().join('');
  const originalDisplay = message.display.bind(message);
  message.display = function () {
    return originalDisplay() + '  ';
  };
  return message;
}

// Step 5: Putting It All Together
function runChatAppDemo() {
  const app = new ChatApp();
  const ui = new MessageListView();
  app.subscribe(ui);

  let msg1 = MessageFactory('text', 'Alice', 'Hello!');
  let msg2 = MessageFactory('image', 'Bob', 'http://image.url/pic.jpg');

  msg1 = emojiDecorator(msg1);
  msg2 = encryptDecorator(msg2);

  app.addMessage(msg1);
  app.addMessage(msg2);
}

runChatAppDemo();
\vspace{1em}
```

16.5.6 Summary

This example shows how combining design patterns helps build modular, maintainable applications:

- The **Singleton** ensures a consistent shared chat state.
- The **Factory** abstracts message creation, allowing easy extension for new types.
- The **Observer** pattern keeps the UI in sync without tight coupling.
- The **Decorator** pattern provides flexible runtime enhancements.

Together, these patterns form a robust foundation for scalable, feature-rich chat apps or similar event-driven applications.

Next, you can extend this app by adding user authentication, message persistence, or real-time networking, all while leveraging these patterns for clean code architecture.

Chapter 17.

Testing Object-Oriented JavaScript

1. Introduction to Unit Testing
2. Testing Classes and Methods
3. Using Jest or Mocha for OOP Code
4. Practical: Writing Tests for a User Management Class

17 Testing Object-Oriented JavaScript

17.1 Introduction to Unit Testing

Unit testing is a fundamental practice in software development where individual pieces of code — typically functions or methods — are tested in isolation to verify that they work as intended. By focusing on the smallest units of functionality, unit tests help catch bugs early, improve code quality, and provide a safety net for future changes.

17.1.1 Why Unit Test?

- **Prevent Regressions:** Unit tests help ensure that changes or new features do not break existing functionality. If a test fails after a code update, it signals a potential regression.
- **Improve Design:** Writing tests often encourages developers to design more modular, testable, and maintainable code. Classes and methods tend to be smaller and more focused when they need to be tested easily.
- **Refactoring Confidence:** With a good suite of tests, developers can confidently refactor or optimize code without fear of silently breaking important behaviors.
- **Documentation:** Tests serve as executable documentation that clearly states how individual parts of the system are expected to behave.

17.1.2 Common Unit Testing Terminology

- **Test Suite:** A collection of related test cases, usually grouped by functionality or by the module/class they test.
- **Test Case:** A single scenario or behavior that is tested, typically represented by a function or method that runs the test.
- **Assertions:** Statements within a test case that verify whether the code under test behaves as expected. For example, asserting that a method returns the correct value.
- **Mocks/Stubs:** Simulated objects or functions used to isolate the unit under test from dependencies, allowing tests to focus solely on the unit's behavior.
- **Test Runner:** A tool or framework that executes tests, reports results, and often provides features like watching files for changes or running tests in parallel.

By mastering unit testing, you not only increase the reliability of your JavaScript OOP code but also foster a development process that is more agile, maintainable, and robust.

17.2 Testing Classes and Methods

Testing object-oriented JavaScript code involves verifying that class constructors, instance methods, and interactions behave as expected. This ensures your classes encapsulate logic correctly, handle edge cases gracefully, and remain robust against changes.

17.2.1 Testing Class Constructors

A constructor initializes class instances with default or provided values. Tests should verify that the constructor:

- Correctly assigns initial values.
- Handles invalid inputs properly (e.g., throws errors or assigns defaults).
- Sets up internal state and dependencies correctly.

Example:

```
class User {
  constructor(name, age) {
    if (!name) throw new Error('Name is required');
    this.name = name;
    this.age = age || 18;
  }
}

// Test case
test('User constructor assigns name and default age', () => {
  const user = new User('Alice');
  expect(user.name).toBe('Alice');
  expect(user.age).toBe(18);
});
\vspace{1em}
```

17.2.2 Testing Instance Methods

Methods define the behavior of your objects. When testing them:

- Provide various inputs and assert expected outputs.
- Verify state changes within the instance.
- Cover both nominal and edge cases.

Example:

```
class Counter {
  constructor() {
    this.count = 0;
  }
  increment() {
```

```
    this.count++;
  }
  reset() {
    this.count = 0;
  }
}

// Test cases
test('Counter increments count', () => {
  const counter = new Counter();
  counter.increment();
  expect(counter.count).toBe(1);
});

test('Counter resets count', () => {
  const counter = new Counter();
  counter.increment();
  counter.reset();
  expect(counter.count).toBe(0);
});
\vspace{1em}
```

17.2.3 Testing Edge Cases

Edge cases reveal bugs hidden under unusual input or usage. When testing classes and methods, consider:

- Invalid or unexpected inputs (e.g., `null`, `undefined`, empty strings).
- Boundary values (e.g., zero, negative numbers, maximum values).
- Repeated calls and state-dependent behavior.

Example:

```
test('User throws error when name is missing', () => {
  expect(() => new User()).toThrow('Name is required');
});
\vspace{1em}
```

17.2.4 Testing Private or Encapsulated Members

JavaScript (ES2022+) supports private fields using `#`, which are not accessible outside the class. For older code, closures may hide internals. Testing private logic can be done by:

- Testing public methods that rely on private state.
- Using reflection or workarounds in non-production environments (discouraged).
- Refactoring complex private logic into testable helper functions (recommended).

Example:

```
class Account {
  #balance = 0;

  deposit(amount) {
    if (amount <= 0) throw new Error('Invalid deposit');
    this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}

test('Account deposits increase balance', () => {
  const acc = new Account();
  acc.deposit(100);
  expect(acc.getBalance()).toBe(100);
});
\vspace{1em}
```

17.2.5 Mocking Dependencies

For classes that depend on external services (e.g., network requests, databases), mocking helps isolate the unit under test:

- Replace dependencies with mock objects/functions.
- Simulate success and failure scenarios.
- Use libraries like Jest to create spies and mocks.

Example:

```
class Logger {
  constructor(output) {
    this.output = output;
  }
  log(message) {
    this.output.write(message);
  }
}

// Test with mock
test('Logger writes message to output', () => {
  const mockOutput = { write: jest.fn() };
  const logger = new Logger(mockOutput);
  logger.log('hello');
  expect(mockOutput.write).toHaveBeenCalledWith('hello');
});
\vspace{1em}
```

By thoroughly testing your classes and methods, you ensure confidence in your codebase, prevent regressions, and encourage well-structured, testable object-oriented design.

17.3 Using Jest or Mocha for OOP Code

As JavaScript applications grow in complexity, using a robust testing framework becomes essential—especially when working with object-oriented code. Two of the most popular testing tools in the JavaScript ecosystem are **Jest** and **Mocha**. Both support testing classes and their behaviors, provide clear syntax for assertions, and streamline the testing process with automation and helpful output.

17.3.1 Introduction to Jest and Mocha

Feature	Jest	Mocha
Built-in test runner	YES Yes	NO No (requires separate runner like `npm test`)
Assertion library	YES Built-in	NO Requires external (e.g., Chai)
Snapshot testing	YES Yes	NO No
Mocking/stubs/spies	YES Built-in	NO Requires third-party (e.g., Sinon)
Setup/teardown hooks	YES Yes	YES Yes
Async testing	YES Yes	YES Yes
Ease of setup	YES Easy (zero-config for many projects)	WARNING More setup required

17.3.2 Setting Up Jest

Install Jest with:

```
npm install --save-dev jest
\vspace{1em}
```

Add a script in your `package.json`:

```
"scripts": {
  "test": "jest"
}
\vspace{1em}
```

Create a file named `calculator.test.js`:

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

```
// Jest test suite
describe('Calculator', () => {
  let calc;

  beforeEach(() => {
    calc = new Calculator();
  });

  test('adds two numbers', () => {
    expect(calc.add(2, 3)).toBe(5);
  });
});
\vspace{1em}
```

Run the tests:

```
npm test
\vspace{1em}
```

17.3.3 Setting Up Mocha Chai

Install Mocha and Chai:

```
npm install --save-dev mocha chai
\vspace{1em}
```

Create a test file `calculator.test.js`:

```
const { expect } = require('chai');

class Calculator {
  add(a, b) {
    return a + b;
  }
}

// Mocha test suite
describe('Calculator', () => {
  let calc;

  beforeEach(() => {
    calc = new Calculator();
  });

  it('should add two numbers', () => {
    expect(calc.add(2, 3)).to.equal(5);
  });
});
\vspace{1em}
```

Add to `package.json`:

```
"scripts": {
  "test": "mocha"
```

```
}
\vspace{1em}
```

Run the tests:

```
npm test
\vspace{1em}
```

17.3.4 Setup and Teardown Hooks

Both Jest and Mocha support lifecycle hooks like:

- `beforeAll` / `before` — Run once before all tests
- `beforeEach` — Run before each test
- `afterEach` — Run after each test
- `afterAll` / `after` — Run once after all tests

Use these to initialize or clean up shared resources.

```
beforeEach(() => {
  // Setup: create a fresh object
});

afterEach(() => {
  // Teardown: reset mocks or states
});
\vspace{1em}
```

17.3.5 Mocking and Spying

Jest (built-in):

```
const logger = {
  log: jest.fn(),
};

logger.log('Hello');

expect(logger.log).toHaveBeenCalledWith('Hello');
\vspace{1em}
```

Mocha Sinon:

```
npm install --save-dev sinon
\vspace{1em}
```

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

```
const logger = {
  log: function (msg) {}
};

const spy = sinon.spy(logger, 'log');
logger.log('Hello');

expect(spy.calledWith('Hello')).to.be.true;
\vspace{1em}
```

17.3.6 Snapshot Testing (Jest Only)

Snapshot tests capture the output of a component (e.g., a stringified object or UI structure) and compare it on subsequent runs.

```
test('snapshot example', () => {
  const user = { name: 'Alice', age: 30 };
  expect(user).toMatchSnapshot();
});
\vspace{1em}
```

17.3.7 Asynchronous Testing

Both Jest and Mocha allow testing async code using:

- Returning a Promise
- `async/await`
- `done()` callback (Mocha)

Jest Example:

```
test('fetches data', async () => {
  const data = await fetchData();
  expect(data).toHaveProperty('user');
});
\vspace{1em}
```

Mocha Example:

```
it('fetches data', async () => {
  const data = await fetchData();
  expect(data).to.have.property('user');
});
\vspace{1em}
```

17.3.8 Conclusion

Both Jest and Mocha are capable of testing object-oriented JavaScript, but Jest stands out with its built-in mocking, snapshot features, and zero-config setup. Mocha remains a flexible and lightweight option, often preferred when combining specific assertion and mocking libraries.

When testing classes, choose the framework that best fits your project's ecosystem and testing needs. Well-tested classes form the foundation of reliable, maintainable object-oriented JavaScript applications.

17.4 Practical: Writing Tests for a User Management Class

In this section, we'll walk through writing unit tests for a `UserManagement` class using a testing framework (we'll use **Jest**, though the concepts apply equally with Mocha/Chai). You'll learn how to test typical class methods like `addUser()`, `removeUser()`, and `findUser()`, while handling both success and failure scenarios.

17.4.1 The `UserManagement` Class

Here's a simple implementation of the class we'll be testing:

```
// userManagement.js
class UserManagement {
  constructor() {
    this.users = [];
  }

  addUser(user) {
    if (!user || !user.id || !user.name) {
      throw new Error('Invalid user data');
    }

    if (this.users.some(u => u.id === user.id)) {
      throw new Error('User already exists');
    }

    this.users.push(user);
    return user;
  }

  removeUser(id) {
    const index = this.users.findIndex(u => u.id === id);
    if (index === -1) {
      throw new Error('User not found');
    }
```

```
    const [removed] = this.users.splice(index, 1);
    return removed;
  }

  findUser(id) {
    return this.users.find(u => u.id === id) || null;
  }
}

module.exports = UserManagement;
\vspace{1em}
```

17.4.2 Writing Unit Tests with Jest

Install Jest (if not already installed):

```
npm install --save-dev jest
\vspace{1em}
```

Add a test script in `package.json`:

```
"scripts": {
  "test": "jest"
}
\vspace{1em}
```

Create a test file: `userManagement.test.js`.

```
// userManagement.test.js
const UserManagement = require('./userManagement');

describe('UserManagement', () => {
  let manager;

  beforeEach(() => {
    manager = new UserManagement();
  });

  // Positive test for addUser
  test('should add a user successfully', () => {
    const user = { id: 1, name: 'Alice' };
    const result = manager.addUser(user);

    expect(result).toEqual(user);
    expect(manager.findUser(1)).toEqual(user);
  });

  // Negative test for addUser (duplicate)
  test('should throw error when adding a duplicate user', () => {
    const user = { id: 1, name: 'Alice' };
    manager.addUser(user);

    expect(() => manager.addUser(user)).toThrow('User already exists');
  });
```

```
  // Negative test for addUser (invalid data)
  test('should throw error on invalid user input', () => {
    expect(() => manager.addUser(null)).toThrow('Invalid user data');
    expect(() => manager.addUser({ name: 'No ID' })).toThrow('Invalid user data');
  });

  // Positive test for removeUser
  test('should remove a user successfully', () => {
    const user = { id: 2, name: 'Bob' };
    manager.addUser(user);

    const removed = manager.removeUser(2);
    expect(removed).toEqual(user);
    expect(manager.findUser(2)).toBeNull();
  });

  // Negative test for removeUser
  test('should throw error when removing non-existent user', () => {
    expect(() => manager.removeUser(999)).toThrow('User not found');
  });

  // Positive test for findUser
  test('should find a user by ID', () => {
    const user = { id: 3, name: 'Charlie' };
    manager.addUser(user);

    const found = manager.findUser(3);
    expect(found).toEqual(user);
  });

  // Test for findUser when not found
  test('should return null for non-existent user', () => {
    expect(manager.findUser(404)).toBeNull();
  });
});
\vspace{1em}
```

17.4.3 Key Testing Practices Highlighted

- **Clear test cases:** Each test targets one behavior and describes its intent clearly.
- **Positive and negative paths:** Tests validate both valid and invalid use cases.
- **Fresh instances per test:** `beforeEach()` ensures no state leaks between tests.
- **Descriptive assertions:** Jest's `expect()` clearly outlines expected outcomes.
- **Error validation:** Throws are validated with meaningful error messages.

17.4.4 Conclusion

Thoroughly testing classes like `UserManagement` ensures reliable, maintainable code. Good tests improve confidence during development and prevent regressions when refactoring. By covering both successful and edge scenarios, your tests become robust tools for verifying and documenting class behavior.

With a well-tested class and suite of unit tests in place, you're ready to confidently scale your object-oriented codebase.

Chapter 18.
OOP in TypeScript

1. TypeScript Classes and Interfaces
2. Strong Typing with OOP
3. Practical: Refactoring a JS Class to TypeScript

18 OOP in TypeScript

18.1 TypeScript Classes and Interfaces

TypeScript is a powerful superset of JavaScript that introduces **strong typing** and advanced features like **interfaces** and **access modifiers** to enhance object-oriented programming. While JavaScript supports class-based OOP, TypeScript elevates it by providing type safety, better tooling, and clearer code contracts.

18.1.1 Benefits of Using TypeScript with OOP

- **Compile-time error checking**: Helps catch bugs early.
- **IntelliSense and autocompletion**: Improves developer productivity in editors like VS Code.
- **Better documentation**: Types make APIs self-descriptive.
- **Enforced contracts**: Interfaces define expected structure for consistency.

18.1.2 TypeScript Class Basics

A class in TypeScript is similar to ES6, but with optional type annotations:

```
class Person {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  greet(): string {
    return `Hello, my name is ${this.name}`;
  }
}

const p = new Person('Alice', 30);
console.log(p.greet()); // Hello, my name is Alice
\vspace{1em}
```

> **Note**: If you omit the type annotations (`: string`, `: number`), TypeScript will infer them where possible, but explicit types are encouraged for clarity.

18.1.3 Access Modifiers

TypeScript adds **access control** keywords to class members:

- `public` (default): Accessible from anywhere.
- `private`: Accessible only within the class.
- `protected`: Accessible within the class and its subclasses.

```
class BankAccount {
  public owner: string;
  private balance: number;

  constructor(owner: string, initialBalance: number) {
    this.owner = owner;
    this.balance = initialBalance;
  }

  public deposit(amount: number): void {
    if (amount > 0) this.balance += amount;
  }

  public getBalance(): number {
    return this.balance;
  }
}

const account = new BankAccount('Bob', 1000);
account.deposit(500);
console.log(account.getBalance()); // 1500
// console.log(account.balance); NO Error: 'balance' is private
\vspace{1em}
```

18.1.4 Inheritance and Subclasses

TypeScript supports classical inheritance with `extends` and allows calling the base class constructor using `super()`:

```
class Employee extends Person {
  constructor(name: string, age: number, public jobTitle: string) {
    super(name, age);
  }

  describe(): string {
    return `${this.name} is a ${this.jobTitle}`;
  }
}
\vspace{1em}
```

18.1.5 Interfaces: Enforcing Contracts

Interfaces define **the shape of objects or classes**, ensuring that any implementing class follows the required structure:

```
interface Drawable {
  draw(): void;
}

class Circle implements Drawable {
  draw(): void {
    console.log('Drawing a circle');
  }
}

class Square implements Drawable {
  draw(): void {
    console.log('Drawing a square');
  }
}

function render(shape: Drawable) {
  shape.draw();
}
\vspace{1em}
```

Interfaces make your code easier to reason about, especially in large teams or APIs.

18.1.6 Interface vs. Type Alias

Both `interface` and `type` can describe object shapes, but `interface` is preferred for class contracts due to its support for extension and implementation.

```
interface Logger {
  log(message: string): void;
}
\vspace{1em}
```

18.1.7 Summary

TypeScript brings structure and safety to object-oriented JavaScript. By adding **static typing**, **interfaces**, and **access modifiers**, it improves code readability, prevents common bugs, and empowers modern tooling. Learning to use TypeScript's OOP features can greatly enhance your code quality and maintainability in large-scale applications.

In the next section, we'll explore how strong typing deepens the advantages of object-oriented design.

18.2 Strong Typing with OOP

TypeScript enhances JavaScript with **strong static typing**, bringing clarity, safety, and maintainability to object-oriented programming. While JavaScript is dynamically typed—meaning types are checked only at runtime—TypeScript enforces types at compile time, reducing bugs and making code easier to understand and refactor.

18.2.1 Why Strong Typing Matters

1. **Catch Errors Early** Types help identify bugs before code runs—like accessing undefined properties or calling methods with wrong parameters.
2. **Improve Readability and Intent** Type annotations serve as built-in documentation, clarifying what a method expects and returns.
3. **Enable Safer Refactoring** Tools like IDEs and linters can safely rename, move, or optimize code when types are known.
4. **Enhance Collaboration** Clear contracts between components improve team coordination and reduce misunderstandings.

18.2.2 Typed Properties and Method Signatures

Here's an example of a class with explicitly typed fields and methods:

```
class User {
  id: number;
  name: string;
  email?: string; // optional

  constructor(id: number, name: string, email?: string) {
    this.id = id;
    this.name = name;
    this.email = email;
  }

  greet(): string {
    return `Hello, ${this.name}`;
  }

  sendEmail(message: string): void {
    if (this.email) {
      console.log(`Sending to ${this.email}: ${message}`);
    }
  }
}
\vspace{1em}
```

Tip: Mark optional properties with `?`, and always define return types for methods.

18.2.3 Interfaces and Contracts

Interfaces are key to strong typing in OOP—they enforce the expected shape of objects or classes.

```
interface Authenticator {
  login(username: string, password: string): boolean;
  logout(): void;
}

class BasicAuth implements Authenticator {
  login(username: string, password: string): boolean {
    // Dummy implementation
    return username === 'admin' && password === '1234';
  }

  logout(): void {
    console.log('Logged out');
  }
}
\vspace{1em}
```

If `BasicAuth` fails to implement any required method, TypeScript raises a compile-time error—helping you uphold architectural contracts.

18.2.4 Impact on Refactoring

Strong typing makes large-scale changes safer:

- IDEs can rename all instances of a variable or method with confidence.
- Type errors appear immediately if something breaks.
- Interfaces provide a clear view of what must change when requirements evolve.

```
// If you change method signature:
logout(reason: string): void;
// You'll get a clear error in every class or object missing this update.
\vspace{1em}
```

18.2.5 Collaboration and Tooling Benefits

Strong typing leads to better developer experiences:

- **Autocomplete**: Editors suggest only valid members and types.
- **Documentation**: You understand objects and functions by reading type annotations.
- **Integration**: APIs and services typed with TypeScript are easier to consume and validate.

18.2.6 Summary

Strong typing in TypeScript's OOP helps enforce consistency, improve collaboration, and catch errors early—especially as codebases scale. Typed classes, interfaces, and method signatures form the foundation for more robust, reliable software development.

Next, we'll apply these principles by refactoring a JavaScript class into fully typed TypeScript.

18.3 Practical: Refactoring a JS Class to TypeScript

One of the most impactful ways to appreciate TypeScript's benefits in object-oriented programming is to refactor an existing JavaScript class into TypeScript. This section walks through the process step by step—adding type annotations, interfaces, and access modifiers—while highlighting how TypeScript improves safety, tooling, and maintainability.

18.3.1 Step 1: Original JavaScript Class

Let's begin with a simple JavaScript `Product` class that manages basic product information.

```
// product.js
class Product {
  constructor(id, name, price) {
    this.id = id;
    this.name = name;
    this.price = price;
  }

  applyDiscount(discountPercent) {
    this.price = this.price - (this.price * discountPercent) / 100;
  }

  getInfo() {
    return `${this.name} - $${this.price.toFixed(2)}`;
  }
}

module.exports = Product;
\vspace{1em}
```

While this works, there are no safeguards to prevent incorrect data or usage. Let's refactor it to TypeScript.

18.3.2 Step 2: Add TypeScript File and Basic Types

Start by creating `Product.ts`. Add type annotations to the constructor parameters and method return types.

```
// Product.ts
class Product {
  id: number;
  name: string;
  price: number;

  constructor(id: number, name: string, price: number) {
    this.id = id;
    this.name = name;
    this.price = price;
  }

  applyDiscount(discountPercent: number): void {
    this.price = this.price - (this.price * discountPercent) / 100;
  }

  getInfo(): string {
    return `${this.name} - $${this.price.toFixed(2)}`;
  }
}
\vspace{1em}
```

> YES Now the compiler will warn you if you accidentally pass a string to `price`, or call `getInfo()` on an undefined object.

18.3.3 Step 3: Add Access Modifiers

Use `public`, `private`, or `protected` to control visibility of class members.

```
class Product {
  private id: number;
  public name: string;
  private price: number;

  constructor(id: number, name: string, price: number) {
    this.id = id;
    this.name = name;
    this.price = price;
  }

  public applyDiscount(discountPercent: number): void {
    this.price = this.price - (this.price * discountPercent) / 100;
```

```
  }

  public getInfo(): string {
    return `${this.name} - $${this.price.toFixed(2)}`;
  }

  public getId(): number {
    return this.id;
  }
}
\vspace{1em}
```

Now `id` and `price` cannot be accessed directly from outside the class, enforcing encapsulation.

18.3.4 Step 4: Define an Interface

Create an interface to define the structure for products. This is especially useful when working with external data (e.g., from APIs).

```
interface IProductData {
  id: number;
  name: string;
  price: number;
}
\vspace{1em}
```

You can use this interface to validate data before creating a `Product`:

```
function createProduct(data: IProductData): Product {
  return new Product(data.id, data.name, data.price);
}
\vspace{1em}
```

18.3.5 Step 5: Type Checking in Action

Now TypeScript helps catch issues early:

```
const badProduct = new Product("A1", "Widget", "19.99"); // NO Error!
\vspace{1em}
```

> Both `id` and `price` must be numbers. The compiler will highlight these type errors immediately.

18.3.6 Step 6: Leverage Tooling and IDE Support

Once refactored to TypeScript:

- Editors offer **autocomplete**, **inline documentation**, and **type checking**.
- Refactoring tools become more reliable (e.g., renaming methods).
- Errors are caught at **compile time**, not during runtime debugging.

18.3.7 Summary

Refactoring JavaScript to TypeScript adds a layer of safety and clarity without changing runtime behavior. By applying:

- **Type annotations** to properties and methods
- **Access modifiers** to encapsulate data
- **Interfaces** to enforce structure

You unlock TypeScript's powerful type system, making object-oriented code more robust and maintainable.

Next, you'll apply these concepts in a TypeScript-based project and explore how types evolve with your application.