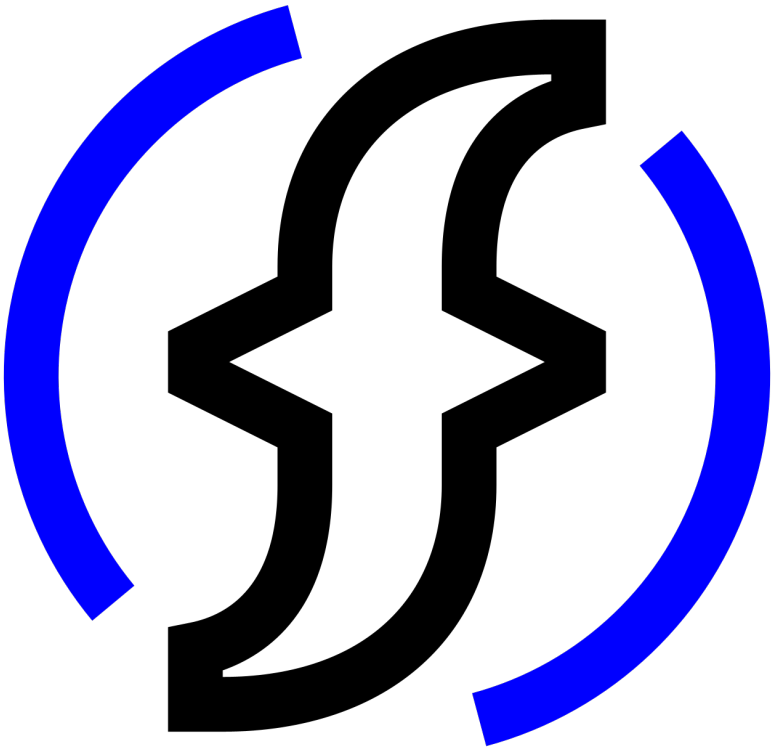
JavaScript Functional Programming



readbytes

JavaScript Functional Programming

From Fundamentals to Advanced Concepts

readbytes.github.io

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Functional Programming

1. What Is Functional Programming?
2. Benefits of Functional Programming in JavaScript
3. Functional vs Imperative: A Quick Comparison
4. Core Principles: Purity, Immutability, First-Class Functions

1 Introduction to Functional Programming

1.1 What Is Functional Programming?

Functional programming (FP) is a **programming paradigm**—a way of thinking about and structuring code—built around **functions** as the central building blocks of software. Unlike procedural or object-oriented programming, which emphasize sequences of commands or objects with internal state, functional programming focuses on **pure functions**, **immutable data**, and **declarative logic** to express computations.

Roots in Mathematics: Lambda Calculus

The foundations of functional programming are deeply rooted in **lambda calculus**, a formal system developed in the 1930s by mathematician Alonzo Church. Lambda calculus treats **functions as first-class citizens** and defines computation entirely through the application of functions to arguments. This idea—simple yet powerful—became a cornerstone of functional programming languages and inspired key concepts used in JavaScript and other modern languages.

In lambda calculus:

- Functions are anonymous (i.e., no name required).
- Functions can be passed around like any other value.
- Computation is based on function application, not variable mutation or side effects.

JavaScript, while not purely functional, supports many of these ideas and allows you to write in a functional style.

How Functional Programming Differs

Paradigm	Focus	Typical Constructs
Procedural	Step-by-step instructions	Loops, variables, conditionals
Object-Oriented	Objects and their interactions	Classes, objects, methods
Functional	Transforming data via functions	Pure functions, higher-order functions

In procedural programming, you describe *how* something should be done. In functional programming, you describe *what* should be done, often with **less code** and **greater clarity**.

Core Idea: Composing Programs with Pure Functions

At its core, functional programming builds programs by composing small, predictable units: **pure functions**. A pure function is one that:

- Given the same input, always returns the same output.
- Has **no side effects** (i.e., it doesn't change any external state).

Instead of mutating data or maintaining hidden state, functional programs work by **transforming data** in a pipeline of operations.

A Simple Example: Doubling Numbers Using `map()`

Let's consider a basic task: doubling a list of numbers.

Imperative style (procedural):

```
const numbers = [1, 2, 3];
const doubled = [];
for (let i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2);
}
console.log(doubled); // [2, 4, 6]
```

Functional style:

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

In the functional version:

- We used the **`map()`** method, which applies a function to each element of the array.
- The function `n => n * 2` is **pure**—it doesn't modify `n`, it just returns a new value.
- No loops, no mutation—just a clear transformation.

This approach scales elegantly. Complex transformations can be composed from smaller, testable functions, resulting in code that is easier to reason about, debug, and maintain.

1.2 Benefits of Functional Programming in JavaScript

Functional programming (FP) offers several **practical advantages** that can make JavaScript code easier to write, understand, test, and maintain. While JavaScript isn't a purely functional language, it includes powerful features—like first-class functions and higher-order methods—that make adopting a functional style both natural and beneficial.

Let's explore the main benefits of functional programming in JavaScript, with clear examples.

Easier Testing and Debugging with Pure Functions

Pure functions don't rely on or change external state—they always return the same output for the same input. This makes them **predictable** and **easy to test** in isolation.

Imperative approach (side effects):

```
let total = 0;
function addToTotal(x) {
  total += x;
}
addToTotal(5);
console.log(total); // 5
```

You have to track external variables (`total`) to understand the behavior.

Functional approach (pure function):

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

No hidden state. No surprises. This function is easy to test because it depends **only** on its inputs.

Improved Readability and Modularity

Functional programming encourages writing **small, focused functions** that do one thing well. These functions can be **composed** together like building blocks, improving code clarity and reuse.

Imperative approach:

```
const users = [
  { name: 'Alice', age: 32 },
  { name: 'Bob', age: 45 }
];

const names = [];
for (let i = 0; i < users.length; i++) {
  if (users[i].age > 40) {
    names.push(users[i].name);
  }
}
console.log(names); // ['Bob']
```

Functional approach:

```
const users = [
  { name: 'Alice', age: 32 },
  { name: 'Bob', age: 45 }
];

const names = users
  .filter(user => user.age > 40)
  .map(user => user.name);

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

The functional version reads like a **sequence of transformations**:

- Filter users by age
- Extract their names

This **declarative style** describes *what* is happening, not *how*.

Safer Code Through Immutability

In functional programming, data is treated as **immutable**—it is never changed in place. Instead, new values are created. This leads to **fewer bugs**, especially in larger applications or when working with shared data.

Imperative (mutable):

```
const person = { name: 'Emma', age: 28 };
person.age = 29; // Mutates the original object
```

Functional (immutable):

```
const person = { name: 'Emma', age: 28 };
const updatedPerson = { ...person, age: 29 }; // Creates a new object
```

By copying and updating data rather than mutating it, we reduce the risk of unexpected side effects elsewhere in the program.

Built-In Support for Higher-Order Functions

JavaScript treats functions as **first-class citizens**, which means they can be:

- Stored in variables
- Passed as arguments
- Returned from other functions

This enables powerful abstractions using **higher-order functions**—functions that take other functions as input or return them as output.

Imperative loop:

```
const numbers = [1, 2, 3, 4];
const evens = [];
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] % 2 === 0) {
    evens.push(numbers[i]);
  }
}
console.log(evens); // [2, 4]
```

Functional with `filter()`:

```
const numbers = [1, 2, 3, 4];
const evens = numbers.filter(n => n % 2 === 0);
console.log(evens); // [2, 4]
```

Using built-in higher-order functions like `map()`, `filter()`, and `reduce()` leads to **shorter, clearer code** that separates *logic* from *control flow*.

1.2.1 Summary

Functional programming brings **clarity**, **predictability**, and **safety** to JavaScript development:

- Pure functions are easier to test and reason about.
- Immutable data reduces bugs caused by shared state.
- Declarative transformations improve code readability.
- Higher-order functions allow elegant, reusable logic.

These benefits scale from small utilities to entire applications—making functional programming a valuable approach in modern JavaScript.

1.3 Functional vs Imperative: A Quick Comparison

One of the best ways to understand **functional programming** is to compare it directly with the more traditional **imperative** style. Both are valid paradigms used in JavaScript, but they approach problems from very different angles.

This section offers a **side-by-side comparison** of the two styles using everyday tasks such as summing an array, filtering data, and updating values. As you read through the examples, take note of how each style manages **state**, controls **flow**, and emphasizes different design philosophies.

Example 1: Summing an Array

Imperative Approach:

```
const numbers = [1, 2, 3, 4, 5];
let total = 0;
for (let i = 0; i < numbers.length; i++) {
  total += numbers[i];
}
console.log(total); // 15
```

Functional Approach:

```
const numbers = [1, 2, 3, 4, 5];
const total = numbers.reduce((sum, n) => sum + n, 0);
console.log(total); // 15
```

What's the difference?

- **Imperative:** Explicit control over every step (looping, accumulating).
- **Functional:** Abstracts away the loop using `reduce()`, focusing on *what* needs to be done—sum the numbers—not *how*.

Example 2: Filtering Items by Condition

Task: Keep only the names starting with the letter "A".

Imperative Approach:

```
const names = ['Alice', 'Bob', 'Angela', 'Mark'];
const aNames = [];
for (let i = 0; i < names.length; i++) {
  if (names[i].startsWith('A')) {
    aNames.push(names[i]);
  }
}
console.log(aNames); // ['Alice', 'Angela']
```

Functional Approach:

```
const names = ['Alice', 'Bob', 'Angela', 'Mark'];
const aNames = names.filter(name => name.startsWith('A'));
console.log(aNames); // ['Alice', 'Angela']
```

What's the difference?

- **Imperative:** Manually builds and mutates a new array.
- **Functional:** Uses `filter()` to express the transformation declaratively.

Example 3: Updating a Value in a List of Objects

Task: Increase the age of each person by 1.

Imperative Approach:

```
const people = [
  { name: 'John', age: 30 },
  { name: 'Jane', age: 25 }
];

for (let i = 0; i < people.length; i++) {
  people[i].age += 1;
}
console.log(JSON.stringify(people,null,2));
// [{ name: 'John', age: 31 }, { name: 'Jane', age: 26 }]
```

Functional Approach (Immutable Update):

```
const people = [
  { name: 'John', age: 30 },
  { name: 'Jane', age: 25 }
];
```

```
const updatedPeople = people.map(person => ({
  ...person,
  age: person.age + 1
}));
console.log(JSON.stringify(updatedPeople,null,2));
// [{ name: 'John', age: 31 }, { name: 'Jane', age: 26 }]
```

What's the difference?

- **Imperative:** Mutates the original array and objects directly.
- **Functional:** Creates a new array with updated copies of the objects, preserving immutability.

1.3.1 Summary: Style, State, and Flow

Feature	Imperative	Functional
Style	Step-by-step instructions	Declarative data transformations
State Management	Often uses mutation	Prefers immutability and stateless functions
Execution Flow	Manual control using loops and logic	Higher-order functions abstract control flow
Readability	Can be verbose and harder to scan	Often more concise and expressive
Testability	Depends on external state	Pure functions are easier to test in isolation

1.3.2 Try This

As a learner, read both styles and ask yourself:

- Which version is easier to follow?
- Which one is safer in terms of avoiding bugs?
- How would each approach scale in a larger application?

There are **trade-offs**. Functional programming often results in more concise and reusable code, but it may initially feel unfamiliar. Imperative code can feel more "direct," but it can become error-prone as complexity grows.

The key is knowing when and how to use each style effectively. In the next section, we'll dive deeper into the core principles that power functional programming—including purity, immutability, and first-class functions.

1.4 Core Principles: Purity, Immutability, First-Class Functions

Functional programming revolves around a few fundamental principles that make your code more **predictable**, **maintainable**, and **easy to reason about**. In JavaScript, these principles guide how you write functions and manage data.

Let's explore the three essential principles:

Pure Functions

A **pure function** is one that:

- Always returns the **same output** given the same input.
- Has **no side effects**—it doesn't modify anything outside its scope (no changing variables, no I/O, no DOM manipulation).

This predictability makes pure functions easy to test and debug.

Example:

```
// Pure function: multiplies input by 2
function double(x) {
  return x * 2;
}

console.log(double(4)); // Always 8
console.log(double(4)); // Always 8 again
```

Contrast with an impure function:

```
let count = 0;
function impureDouble(x) {
  count++;              // Side effect: modifies external variable
  return x * 2 + count;
}

console.log(impureDouble(4)); // 9 (8 + 1)
console.log(impureDouble(4)); // 10 (8 + 2) - different result for same input!
```

Pure functions keep your code **deterministic** and side-effect free.

Immutability

Immutability means data **cannot be changed** once created. Instead, you create **new copies** of data with the desired changes.

Immutability prevents bugs caused by accidental changes to shared data and makes state management safer and clearer.

Example:

```
const person = { name: 'Anna', age: 25 };
```

```
// Immutable update: create a new object with updated age
const olderPerson = { ...person, age: 26 };

console.log(person.age);      // 25 (original unchanged)
console.log(olderPerson.age); // 26 (new object)
```

In contrast, mutating the original object directly:

```
person.age = 26;
console.log(person.age); // 26 (original changed)
```

Using immutability helps avoid unintended side effects and makes your data flow easier to track.

First-Class Functions

In JavaScript, **functions are first-class citizens**, meaning they can be:

- Stored in variables,
- Passed as arguments to other functions,
- Returned from functions.

This allows you to write **higher-order functions** and build flexible, reusable abstractions.

Example:

```
// Function stored in a variable
const greet = name => `Hello, ${name}!`;

// Function passed as an argument
function sayHello(fn, name) {
  console.log(fn(name));
}

sayHello(greet, 'Sam'); // Hello, Sam!

// Function returned from a function
function multiplier(factor) {
  return number => number * factor;
}

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

First-class functions empower you to write concise, expressive code that composes behavior dynamically.

1.4.1 Why These Principles Matter

Together, **pure functions**, **immutability**, and **first-class functions** form the backbone of functional programming:

- They make your code **predictable** by eliminating hidden state and side effects.
- They improve **maintainability** by encouraging small, reusable functions and immutable data.
- They enable powerful **abstractions** through functions as values.

By embracing these principles in JavaScript, you'll write cleaner, more robust programs that scale well as your projects grow.

Next up, we will see these principles in action through practical examples and deeper explorations in upcoming chapters!

Chapter 2.

JavaScript as a Functional Language

1. JavaScript Function Fundamentals
2. Functions as First-Class Citizens
3. Higher-Order Functions
4. Anonymous and Arrow Functions

2 JavaScript as a Functional Language

2.1 JavaScript Function Fundamentals

Functions are at the heart of JavaScript—they're one of the core language features that make it incredibly flexible and powerful. In this section, we'll review the basics of how functions work in JavaScript, covering:

- Function declarations and expressions
- Parameters and return values
- Function scope
- Treating functions as values

This foundation is essential as you move towards more advanced functional programming concepts.

Function Declarations

A **function declaration** defines a named function using the `function` keyword. These functions are hoisted, meaning they are available anywhere in their scope, even before the line where they're declared.

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

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

- `greet` is the function name.
- `name` is the parameter.
- The function returns a greeting string.

Function Expressions

Functions can also be defined as **expressions** and assigned to variables. Unlike declarations, function expressions are not hoisted.

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

console.log(add(2, 3)); // 5
```

Parameters and Return Values

Functions receive **parameters** as inputs and can **return** values. If no `return` statement is present, the function returns `undefined` by default.

```
function square(x) {
  return x * x;
```

```
}

console.log(square(4)); // 16
```

Scope

Functions create their own **scope**, meaning variables declared inside a function are not accessible outside it.

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

sayHello();       // Hi!
console.log(message); // ReferenceError: message is not defined
```

Understanding scope helps prevent variable conflicts and supports modular code.

Functions as Values

In JavaScript, functions are **first-class values**. This means:

- You can assign functions to variables.
- Pass functions as arguments.
- Return functions from other functions.

This ability allows you to write highly reusable and composable code.

```
const multiply = (x, y) => x * y;

function operate(fn, a, b) {
  return fn(a, b);
}

console.log(operate(multiply, 5, 6)); // 30
```

Here, `multiply` is passed as a value to the `operate` function, which calls it dynamically.

2.1.1 Summary

- Functions in JavaScript can be declared or expressed.
- They take parameters, can return values, and create local scopes.
- Most importantly, functions themselves are values you can manipulate.

Mastering these fundamentals sets the stage for exploring JavaScript's powerful functional programming capabilities in the upcoming sections.

2.2 Functions as First-Class Citizens

In JavaScript, functions are **first-class citizens**, which means they are treated like any other value in the language. You can:

- Store functions in variables,
- Pass them as arguments to other functions,
- Return them from functions.

This flexibility is a powerful feature that enables many functional programming techniques and patterns.

Storing Functions in Variables

Just like numbers, strings, or objects, you can assign a function to a variable. This allows you to store behavior in a variable and call it later.

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

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

Here, `greet` holds a function that can be invoked just like any other function.

Passing Functions as Arguments

Functions can be passed as parameters to other functions. This enables **callbacks**, where one function controls when and how another function executes.

A classic example is `setTimeout`, which takes a function to run after a delay:

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

Or using an arrow function for brevity:

```
setTimeout(() => console.log('Hello after 2 seconds'), 2000);
```

Passing functions like this allows asynchronous or deferred execution.

Returning Functions from Functions

Functions can also **return other functions**, enabling dynamic behavior and function factories.

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

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

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

Here, `multiplier` returns a new function that remembers the `factor` and applies it when called.

Functions as Object Properties

Functions can be stored as properties on objects, acting as methods.

```
const calculator = {
  add: function(a, b) {
    return a + b;
  },
  subtract(a, b) {
    return a - b;
  }
};

console.log(calculator.add(4, 3));      // 7
console.log(calculator.subtract(4, 3)); // 1
```

This demonstrates that functions are flexible values that can be organized and used in many ways.

2.2.1 Why This Matters

Treating functions as first-class citizens gives you the ability to:

- Abstract and encapsulate behavior,
- Create higher-order functions (functions that operate on other functions),
- Build composable and reusable code structures.

This foundation is key to mastering functional programming in JavaScript and will be explored further in the next sections.

2.3 Higher-Order Functions

A **higher-order function** (HOF) is a function that **either takes one or more functions as arguments, returns a function, or both**. This concept is central to functional programming and allows you to write flexible, reusable code that can abstract control flow, transform data, or extend behavior dynamically.

Built-In Higher-Order Functions in JavaScript

JavaScript provides many built-in higher-order functions for working with arrays. Some of the most commonly used are `map()`, `filter()`, and `reduce()`.

`map()`

- Takes a function as an argument.
- Applies that function to every item in an array.
- Returns a new array with the transformed values.

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

`filter()`

- Takes a function (predicate) as an argument.
- Returns a new array containing only elements for which the function returns `true`.

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter(n => n % 2 === 0);
console.log(evens); // [2, 4]
```

`reduce()`

- Takes a function and an initial value.
- Combines all elements of an array into a single value by repeatedly applying the function.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, n) => acc + n, 0);
console.log(sum); // 10
```

These built-in methods show how higher-order functions let you describe **what** you want to do with data instead of **how** to do it with loops.

Creating a Custom Higher-Order Function

You can also create your own higher-order functions. For example, here's a simple **logging wrapper** that takes a function and returns a new function that logs inputs and outputs:

```
function withLogging(fn) {
  return function(...args) {
    console.log('Calling function with arguments:', args);
    const result = fn(...args);
    console.log('Function returned:', result);
    return result;
  };
}
```

```javascript
// A simple function to multiply two numbers
function multiply(a, b) {
  return a * b;
}

// Wrap multiply with logging
const loggedMultiply = withLogging(multiply);

loggedMultiply(3, 4);
// Output:
// Calling function with arguments: [3, 4]
// Function returned: 12
```

Here, `withLogging` is a higher-order function because it **returns a new function** that enhances the behavior of the original.

2.3.1 Summary

- Higher-order functions either **take functions as arguments** or **return functions**.
- Built-in HOFs like `map()`, `filter()`, and `reduce()` simplify working with collections declaratively.
- Custom HOFs let you abstract patterns like logging, caching, or retries by wrapping existing functions.

Understanding and using higher-order functions is a key step toward writing clean, reusable, and expressive functional JavaScript code.

2.4 Anonymous and Arrow Functions

In JavaScript, functions come in several forms—**named functions**, **anonymous functions**, and **arrow functions**—each with its own syntax, behavior, and best use cases. Understanding the differences helps you write clearer, more effective functional code.

Named Functions

Named functions have an explicit name, which helps with debugging and recursion. They can be declared using the `function` keyword:

```javascript
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}

console.log(factorial(5)); // 120
```

- Useful when a function calls itself (recursion).
- The name shows up in stack traces, aiding debugging.
- Can be **hoisted** (available before declaration in code).

Anonymous Functions

Anonymous functions are functions without a name. They are often assigned to variables or passed as arguments.

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

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

- Typically used when the function is used as a one-off or passed immediately.
- Less verbose than named functions but harder to debug (name won't appear in stack traces).
- Not hoisted; accessible only after assignment.

Arrow Functions

Introduced in ES6, **arrow functions** offer a concise syntax and have special behavior regarding `this`.

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

- Syntax is shorter, especially for simple functions.
- Implicit return when using expression bodies (no need for `return` keyword).
- **Do not have their own `this` context**—they inherit `this` from the surrounding scope.
- Cannot be used as constructors (`new` operator).
- No `arguments` object inside arrow functions.

`this` Behavior: Function vs Arrow

Named and anonymous functions define their own `this` based on how they are called:

```
const person = {
  age: 30,
  growOlder: function() {
    this.age++;
  }
};
person.growOlder();
console.log(person.age); // 31
```

Arrow functions capture `this` from their surrounding context:

```
const person = {
  age: 30,
  growOlder: () => {
    this.age++;
  }
};
person.growOlder();
console.log(person.age); // 30 (does NOT work as expected)
```

Because arrow functions do not bind their own `this`, they're great inside callbacks to preserve context, but **not suitable** for object methods that rely on `this`.

When to Use Which?

Use Case	Recommended Function Type
Simple one-liners or callbacks	Arrow functions
Functions needing recursion	Named functions
Methods that use `this`	Named or anonymous functions
Passing functions inline (e.g., `map()`, `filter()`)	Arrow functions

Examples: Arrow Functions in Action

Using arrow functions inside `map()` for clean, readable code:

```
const numbers = [1, 2, 3];
const squares = numbers.map(n => n * n);
console.log(squares); // [1, 4, 9]
```

Using a named function for recursion improves readability:

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(6)); // 8
```

2.4.1 Summary

- **Named functions** have names, support recursion, and are hoisted.
- **Anonymous functions** are unnamed and typically used as values or callbacks.
- **Arrow functions** offer concise syntax and lexically bind `this`, ideal for inline callbacks.
- Choosing the right function type depends on your use case, especially around `this` and readability.

Mastering these different function types will help you write cleaner, more effective JavaScript

functional code.

Chapter 3.

Immutability and Side Effects

1. Mutable vs Immutable Values
2. Avoiding Side Effects
3. Using `const`, `Object.freeze()`, and Shallow/Deep Copies
4. Practical Example: Immutable State in a Counter

3 Immutability and Side Effects

3.1 Mutable vs Immutable Values

Understanding **mutable** and **immutable** values is fundamental to writing predictable and bug-resistant JavaScript, especially when adopting a functional programming style.

What Are Mutable and Immutable Values?

- **Mutable values** can be changed or modified **after** they are created.
- **Immutable values** cannot be changed once created; instead, you create **new copies** with the desired changes.

In JavaScript, **primitive types** like numbers, strings, and booleans are **immutable**. However, **objects and arrays** are **mutable** by default because they are reference types.

Mutation with Arrays and Objects

When you modify an array or object directly, you are performing a **mutation**. This changes the original data and can lead to bugs, especially when other parts of the program hold references to the same data.

Example: Mutating an array

```
const fruits = ['apple', 'banana'];
fruits.push('orange'); // Mutates original array
console.log(fruits);   // ['apple', 'banana', 'orange']
```

The `push()` method modifies the original `fruits` array. If other code also references this array, it now sees the updated version—sometimes unexpectedly.

Example: Mutating an object

```
const user = { name: 'Alice', age: 25 };
user.age = 26; // Direct mutation
console.log(JSON.stringify(user,null,2)); // { name: 'Alice', age: 26 }
```

The property `age` is changed in place.

Functional Alternatives: Avoiding Mutation

To keep data **immutable**, you avoid changing the original object or array. Instead, you create **new copies** with the updates applied.

For arrays, instead of `push()` or `splice()`:

- Use the **spread operator** `(...)` to create new arrays:

```
const fruits = ['apple', 'banana'];
const newFruits = [...fruits, 'orange']; // New array, original untouched
console.log(newFruits); // ['apple', 'banana', 'orange']
```

```
console.log(fruits);    // ['apple', 'banana']
```

- Use **`concat()`** to add elements without mutation:

```
const fruits = ['apple', 'banana'];
const newFruits = fruits.concat('orange');
console.log(newFruits); // ['apple', 'banana', 'orange']
```

- Use **`map()`** to transform arrays without mutation:

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

For objects, instead of modifying properties directly:

- Use the **spread operator** to create a new object with updated properties:

```
const user = { name: 'Alice', age: 25 };
const updatedUser = { ...user, age: 26 };
console.log(JSON.stringify(updatedUser,null,2)); // { name: 'Alice', age: 26 }
console.log(JSON.stringify(user,null,2));        // { name: 'Alice', age: 25 }
```

3.1.1 Why Choose Immutability?

- **Predictability:** Data doesn't change unexpectedly.
- **Easier debugging:** You can trace changes through new versions of data.
- **Safe sharing:** Multiple parts of your code can safely reference the same data without interfering.

3.1.2 Summary

- **Mutable values** like arrays and objects can be changed in place.
- Direct mutations lead to side effects that are hard to track.
- Using **immutable patterns** like spreading, `map()`, and `concat()` creates new versions without altering originals.
- Embracing immutability is a key step towards writing cleaner, safer functional JavaScript code.

3.2 Avoiding Side Effects

What Is a Side Effect?

In programming, a **side effect** occurs when a function or expression modifies some state or interacts with the outside world **beyond returning a value**. This includes:

- Changing variables or data outside the function's scope (like global variables),
- Modifying objects or arrays passed by reference,
- Performing input/output operations such as logging to the console or writing to a file,
- Triggering network requests or database updates.

Real-world examples:

```
let count = 0;

function increment() {
  count++;                    // Side effect: modifies external variable
}

function logMessage(msg) {
  console.log(msg);           // Side effect: interacts with the console
}
```

Both functions do more than just return a value—they change or affect something outside their local environment.

Why Avoid Side Effects in Functional Programming?

Functional programming strives for **predictability** and **reliability** by minimizing or eliminating side effects. Here's why avoiding side effects matters:

- **Easier to reason about:** Functions that depend only on their inputs and produce outputs without changing outside state are **deterministic**—always producing the same output for the same input.
- **Simpler testing:** Pure functions can be tested in isolation without worrying about the broader program context.
- **Improved maintainability:** Code without hidden side effects is less prone to bugs caused by unexpected changes to shared data.
- **Better concurrency:** Without side effects, functions can safely run in parallel or be reused without risk of conflicts.

Writing Pure Functions: No Side Effects Allowed

A **pure function** follows two rules:

1. Given the same inputs, it always returns the same output.
2. It does not cause any observable side effects outside the function.

Example of a pure function:

```
function square(x) {
  return x * x;  // Uses only input, no external interaction
}

console.log(square(4)); // 16
console.log(square(4)); // 16 again - always the same
```

No variables outside the function are modified, and it doesn't perform any I/O or state changes.

Side Effect vs Pure Function: Example

Impure function with side effect:

```
let total = 0;

function addToTotal(amount) {
  total += amount;      // Mutates external state
}
```

Pure alternative without side effect:

```
function add(a, b) {
  return a + b;        // Returns new value without changing external state
}

const total = 0;
const newTotal = add(total, 5);
console.log(newTotal); // 5
console.log(total);    // 0 (original unchanged)
```

3.2.1 Summary

- **Side effects** occur when functions modify external state or interact with the environment.
- Avoiding side effects leads to **pure functions** that are easier to test, debug, and reason about.
- Writing functions that rely **only on their inputs** and produce outputs without changing anything else is a cornerstone of functional programming.

In the next section, we'll explore JavaScript tools like `const`, `Object.freeze()`, and copying techniques that help manage immutability and further reduce side effects.

3.3 Using `const`, `Object.freeze()`, and Shallow/Deep Copies

In JavaScript, managing immutability involves several tools and techniques. This section explores how to use `const`, `Object.freeze()`, and copying methods to help create immutable data structures.

Using `const` to Prevent Variable Reassignment

The `const` keyword declares a variable that **cannot be reassigned** to a new value. However, for objects and arrays, `const` **does not make the contents immutable**—it only prevents the variable from pointing to a different reference.

```
const numbers = [1, 2, 3];
numbers.push(4);      // Allowed: modifies the array content
console.log(numbers); // [1, 2, 3, 4]

// numbers = [5, 6];  // Error: Assignment to constant variable
```

Key takeaway: `const` protects the variable binding but **not** the object's internal data.

Using `Object.freeze()` for Shallow Immutability

`Object.freeze()` prevents changes to an object's **top-level properties**—making it **read-only**. However, this is a **shallow freeze**, meaning nested objects can still be modified.

```
const user = Object.freeze({
  name: 'Alice',
  address: {
    city: 'Toronto'
  }
});

user.name = 'Bob';              // Ignored silently or throws error in strict mode
console.log(user.name);         // 'Alice'

// But nested objects are still mutable:
user.address.city = 'Montreal';
console.log(user.address.city); // 'Montreal'
```

Use `Object.freeze()` when you want to prevent accidental changes to the top-level object, especially in small or flat data structures.

Deep Copies and Deep Freezing

When your data contains **nested objects or arrays**, a shallow freeze or copy isn't enough to ensure full immutability.

- To create a **deep copy** of an object or array (a new object with all nested values copied), you can use:

  ```
  // Modern JavaScript: structuredClone (supported in many environments)
  const original = { a: 1, b: { c: 2 } };
  const deepCopy = structuredClone(original);
  ```

```
deepCopy.b.c = 3;
console.log(original.b.c); // 2 (unchanged)
```

- For older environments, custom deep copy functions or libraries like **Lodash's `cloneDeep`** can be used.
- To create a **deep freeze** (freeze all nested objects), you can write a recursive function:

```
function deepFreeze(obj) {
  Object.freeze(obj);
  Object.values(obj).forEach(value => {
    if (value && typeof value === 'object') {
      deepFreeze(value);
    }
  });
  return obj;
}
```

Creating Immutable Updates with Spread Syntax and Destructuring

Instead of mutating arrays or objects directly, use **spread syntax** to create new copies with the desired updates:

Immutable array update:

```
const fruits = ['apple', 'banana'];
const newFruits = [...fruits, 'orange'];
console.log(newFruits); // ['apple', 'banana', 'orange']
console.log(fruits);    // ['apple', 'banana'] (original unchanged)
```

Immutable object update:

```
const user = { name: 'Alice', age: 25 };
const updatedUser = { ...user, age: 26 };
console.log(updatedUser); // { name: 'Alice', age: 26 }
console.log(user);        // { name: 'Alice', age: 25 } (original unchanged)
```

3.3.1 Summary

- `const` prevents variable reassignment but **does not** make objects or arrays immutable.
- `Object.freeze()` creates a **shallow read-only** object; nested objects remain mutable.
- Use **deep copy** methods like `structuredClone()` or custom functions to fully clone nested data.
- Spread syntax and destructuring allow creating **new immutable copies** with updates.

These techniques are essential for managing immutable data structures in JavaScript, helping you write safer, side-effect-free functional code.

3.4 Practical Example: Immutable State in a Counter

Managing state immutably is a common pattern in functional programming. Let's explore this concept with a simple **counter** example.

Mutable Counter (With Object Mutation)

Here, the counter state is an object that gets **mutated directly** when incrementing or decrementing:

```
const counter = { count: 0 };

function increment(state) {
  state.count += 1;  // Mutates original object
  return state;
}

function decrement(state) {
  state.count -= 1;  // Mutates original object
  return state;
}

console.log(JSON.stringify(increment(counter),null,2)); // { count: 1 }
console.log(JSON.stringify(decrement(counter),null,2)); // { count: 0 }
console.log(JSON.stringify(counter,null,2));             // { count: 0 } - original mutated
```

Issue: The original `counter` object is changed, which can lead to bugs if other parts of the code rely on the old state.

Immutable Counter (Using Pure Functions)

To avoid mutation, we create new state objects for every update. The original state remains untouched.

```
const counter = { count: 0 };

function increment(state) {
  // Return a new object with count incremented
  return { ...state, count: state.count + 1 };
}

function decrement(state) {
  // Return a new object with count decremented
  return { ...state, count: state.count - 1 };
}

const state1 = increment(counter);
console.log(JSON.stringify(state1,null,2));     // { count: 1 }
console.log(JSON.stringify(counter,null,2));    // { count: 0 } - original unchanged

const state2 = decrement(state1);
console.log(JSON.stringify(state2,null,2));     // { count: 0 }
console.log(JSON.stringify(state1,null,2));     // { count: 1 } - previous state unchanged
```

Alternative: Using `Object.assign()`

If you prefer, you can also use `Object.assign()` to create new state objects immutably:

```
function increment(state) {
  return Object.assign({}, state, { count: state.count + 1 });
}

function decrement(state) {
  return Object.assign({}, state, { count: state.count - 1 });
}
```

Both spread syntax (`{ ...state, ... }`) and `Object.assign()` achieve the same goal: creating new objects instead of mutating the original.

3.4.1 Why Use This Immutable Pattern?

- **Predictability:** Each state update produces a new state without side effects.
- **History tracking:** You can keep previous states intact (useful for undo features).
- **Better debugging:** You can inspect state changes step-by-step without unexpected mutations.

3.4.2 Summary

This simple example demonstrates how **pure functions** and **immutable updates** work hand in hand to make state management safer and more predictable. By returning new state objects in `increment()` and `decrement()`, your counter behaves in a truly functional style.

In the next chapter, we will build on this concept to manage more complex application state immutably.

Chapter 4.

Pure Functions

1. What Makes a Function Pure?
2. Determinism and Referential Transparency
3. Identifying and Refactoring Impure Code

4 Pure Functions

4.1 What Makes a Function Pure?

In functional programming, a **pure function** is the foundation for writing predictable and maintainable code. But what exactly makes a function *pure*?

Definition of a Pure Function

A function is considered **pure** if it meets two essential criteria:

1. **Deterministic output:** For the **same inputs**, it always returns the **same output**.
2. **No side effects:** It does **not modify anything outside its scope** or interact with the outside world (no changing variables, no I/O, no state mutation).

Why These Criteria Matter

- **Same input → Same output:** This guarantees that the function's behavior is consistent and predictable, making it easier to understand and reason about.
- **No side effects:** The function only computes and returns a value without affecting anything else, which avoids hidden bugs caused by unexpected external changes.

Together, these criteria make pure functions **referentially transparent**—you can replace the function call with its output value anywhere without changing the program's behavior.

Examples of Pure and Impure Functions

Pure function example:

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

console.log(add(2, 3)); // 5
console.log(add(2, 3)); // 5 (always the same output)
```

- Always returns the same result for inputs `(2, 3)`.
- Does not modify any external state or variables.

Impure function examples:

1. Function with side effect (modifies external variable):

```
let counter = 0;

function increment() {
  counter += 1; // Modifies external state (side effect)
  return counter;
}

console.log(increment()); // 1
console.log(increment()); // 2 (different output for no input)
```

- The output depends on external variable `counter`.
- Calling it twice with the same inputs (none in this case) produces different results.

2. Function with side effect (console logging):

```
function greet(name) {
  console.log(`Hello, ${name}!`); // Side effect: output to console
  return `Hello, ${name}!`;
}
```

- Even though it returns a predictable string, it performs a side effect by logging, so it's impure.

Why Purity Matters

- **Predictability:** Pure functions make your code's behavior easier to anticipate and understand.
- **Testability:** Since pure functions depend only on inputs, writing automated tests becomes straightforward.
- **Debugging:** Pure functions reduce hidden bugs caused by shared or mutable state.
- **Concurrency:** Pure functions can run in parallel or be memoized safely, improving performance and scalability.

4.1.1 Summary

- A **pure function** always returns the same output for the same inputs and causes **no side effects**.
- Impure functions may change external state or produce different outputs despite identical inputs.
- Purity improves predictability, testability, and overall code quality, making it a cornerstone of functional programming.

In the next section, we will dive deeper into **determinism** and **referential transparency**, two properties tightly linked to function purity.

4.2 Determinism and Referential Transparency

Two foundational concepts underpin pure functions: **determinism** and **referential transparency**. Understanding these ideas deepens your grasp of what makes functions predictable, testable, and optimizable.

Determinism: Output Depends Only on Input

A function is **deterministic** if it always produces the same output when given the same input — nothing else influences its result.

Example of determinism:

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

console.log(multiply(2, 3)); // 6
console.log(multiply(2, 3)); // 6 again - always the same output
```

No matter how many times or where you call `multiply(2, 3)`, the output is always `6`.

Non-deterministic example:

```
function randomNumber() {
  return Math.random();
}

console.log(randomNumber()); // e.g., 0.43
console.log(randomNumber()); // e.g., 0.79 (different every time)
```

`randomNumber` is not deterministic because it does not depend on input and produces different results each time.

Referential Transparency: Expressions Can Be Replaced by Their Values

An expression is **referentially transparent** if you can replace it with its corresponding value without changing the behavior of the program.

This means you can substitute a function call with its output value safely.

Example:

```
const result = multiply(4, 5); // result = 20
const area = result * 10;      // area = 200

// Because multiply is pure and deterministic, we could replace it inline:
const areaDirect = 20 * 10;    // Same as above, no difference in behavior
```

Here, the expression `multiply(4, 5)` can be replaced by `20` without altering the meaning of the program.

Why is this useful? It enables:

- **Simplified reasoning:** You understand the program's flow better when you can substitute expressions with values.
- **Code optimization:** Compilers or JavaScript engines can cache (memoize) results or eliminate redundant calls.
- **Refactoring safety:** Changing or rearranging code without unintended side effects.

Referential Transparency in Code: Before and After Replacement

```
function square(x) {
  return x * x;
}

const y = square(3) + square(3);  // Calls square twice
// Because square is pure, this is equivalent to:
const value = 9 + 9;
```

Since `square(3)` always returns `9`, both expressions behave identically.

4.2.1 Summary

- **Determinism** means a function's output depends solely on its inputs and is always the same for those inputs.
- **Referential transparency** means expressions can be replaced with their computed values without affecting the program's behavior.
- Together, they allow for easier reasoning, testing, and optimization of code.
- Pure functions are deterministic and referentially transparent, making them reliable building blocks in functional programming.

Next, we will learn how to identify impure functions and transform them into pure ones to improve your code quality.

4.3 Identifying and Refactoring Impure Code

To write better functional code, it's crucial to **identify impure functions** and learn how to refactor them into **pure functions**. This section walks you through spotting impurities and fixing them step-by-step.

How to Spot Impure Functions

Impure functions often:

- Modify **external variables or objects** (side effects).
- Perform **input/output** operations like logging or API calls.
- Depend on **external or global state** that's not passed in as arguments.
- Produce **different outputs** for the same input.

Example 1: Impure Function Modifying External State

```
let total = 0;

function addToTotal(amount) {
  total += amount;  // Side effect: modifies external variable
  return total;
}

console.log(addToTotal(5));  // 5
console.log(addToTotal(3));  // 8 (output depends on external state)
```

Problems:

- The function changes `total` outside its scope.
- Calling it repeatedly changes behavior, breaking predictability.

Refactoring Into a Pure Function

Instead of relying on external `total`, pass the current state as an argument and return a new value:

```
function add(currentTotal, amount) {
  return currentTotal + amount; // No side effect
}

const total1 = add(0, 5);
const total2 = add(total1, 3);

console.log(total1); // 5
console.log(total2); // 8
```

Now, the function depends solely on inputs and returns a new result without modifying external data.

Example 2: Impure Function with Console Logging

```
function greet(name) {
  console.log(`Hello, ${name}!`);  // Side effect: logs to console
  return `Hello, ${name}!`;
}
```

Logging inside the function is a side effect, so the function is impure.

Refactoring to Separate Side Effects

Split the pure logic and side effects:

```
function greet(name) {
  return `Hello, ${name}!`;  // Pure: returns string based on input
}
```

```
const message = greet('Alice');
console.log(message);          // Side effect happens outside
```

This approach keeps `greet` pure and confines side effects to the caller.

Example 3: Mutating an Object Inside a Function

```
function updateUserName(user, newName) {
  user.name = newName;  // Mutation of external object
  return user;
}

const user = { name: 'Bob' };
updateUserName(user, 'Alice');

console.log(user.name); // 'Alice' (original object mutated)
```

Refactoring with Immutable Updates

Return a new object instead of mutating the input:

```
function updateUserName(user, newName) {
  return { ...user, name: newName };  // Returns a new object
}

const user = { name: 'Bob' };
const updatedUser = updateUserName(user, 'Alice');

console.log(user.name);         // 'Bob' (original unchanged)
console.log(updatedUser.name); // 'Alice'
```

4.3.1 Summary: Refactoring Steps

1. **Identify side effects:** Look for external state modifications, I/O, or dependencies beyond input parameters.
2. **Pass all necessary data explicitly:** Avoid relying on globals or outside variables.
3. **Return new values:** Don't mutate inputs; create and return new objects or primitives.
4. **Separate side effects:** Keep pure logic and effects like logging or network calls separate.

By practicing these techniques, you'll transform impure, unpredictable functions into clean, reusable, and testable pure functions — the hallmark of functional programming in JavaScript.

Chapter 5.

Higher-Order Functions

1. Functions as Arguments
2. Functions as Return Values
3. Practical Patterns: Function Transformers and Wrappers

5 Higher-Order Functions

5.1 Functions as Arguments

One of the most powerful features of JavaScript—and functional programming in general—is the ability to **pass functions as arguments** to other functions. This allows you to write **flexible, reusable, and expressive code** that can be customized by supplying different behaviors.

What Does It Mean to Pass a Function as an Argument?

In JavaScript, functions are **first-class citizens**, meaning they can be:

- Stored in variables,
- Passed as arguments to other functions,
- Returned from functions.

When you pass a function as an argument, the receiving function can **call it whenever needed**, allowing you to customize how it behaves.

Why Pass Functions as Arguments?

- **Reusability:** Write a generic function once and customize behavior by passing different functions.
- **Expressiveness:** Code reads more declaratively by describing *what* should be done rather than *how*.
- **Separation of concerns:** Different parts of logic stay separate, improving modularity and maintainability.

Built-in Examples: `map()` and `filter()`

JavaScript arrays come with several built-in higher-order functions that take other functions as arguments.

- **`map()`** transforms each element of an array using a function.
- **`filter()`** selects elements based on a condition function.

Example: Using `map()` to double numbers

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

Or with arrow functions for brevity:

```
const doubled = numbers.map(n => n * 2);
```

Example: Using `filter()` to select even numbers

```
const numbers = [1, 2, 3, 4, 5, 6];
const evens = numbers.filter(function (n) {
  return n % 2 === 0;
});
console.log(evens); // [2, 4, 6]
```

Or using arrow function syntax:

```
const evens = numbers.filter(n => n % 2 === 0);
```

Small Custom Example: Filtering Names by Length

```
const names = ['Alice', 'Bob', 'Charlie', 'Dave'];

function isLongName(name) {
  return name.length > 3;
}

const longNames = names.filter(isLongName);
console.log(longNames); // ['Alice', 'Charlie', 'Dave']
```

Here, the `filter` function accepts `isLongName` as a callback, which defines the filtering logic separately from the filtering mechanism.

5.1.1 Summary

- Passing functions as arguments lets you customize behavior dynamically.
- This pattern powers many built-in methods like `map()` and `filter()`, enabling concise, expressive code.
- Writing functions that accept other functions increases **flexibility**, **reusability**, and **clarity** in your programs.

Next, we will explore how functions can also be **returned** from other functions, adding even more power to your functional programming toolkit.

5.2 Functions as Return Values

Another powerful concept in JavaScript functional programming is that **functions can return other functions**. This enables the creation of highly **customizable** and **composable** utilities that can be tailored on the fly.

What Does It Mean for a Function to Return a Function?

A function that returns another function is sometimes called a **higher-order function**. The returned function can then be invoked later, possibly with additional arguments.

This pattern allows you to build **function factories**—functions that create specialized functions based on input parameters.

Why Return Functions?

- **Customization:** You create general-purpose functions that generate specific behaviors.
- **Composition:** Returned functions can be combined or chained for complex logic.
- **Partial application:** You can fix some arguments upfront and get a function waiting for the rest.

Example 1: Function Factory Multiplier Generator

Imagine you want a function that creates multipliers:

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

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

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

Here, `createMultiplier` returns a new function configured with the specified `factor`. This returned function remembers the `factor` via **closure** and multiplies any given number by it.

Example 2: Currying Returning Partially Applied Functions

Currying is the process of transforming a function with multiple arguments into a sequence of functions each taking a single argument.

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

const addFive = add(5);
console.log(addFive(10)); // 15
```

Instead of calling `add(5, 10)`, the curried version returns a function waiting for the second argument.

Arrow Function Syntax for Brevity

Using arrow functions, the multiplier example becomes:

```
const createMultiplier = factor => number => number * factor;

const double = createMultiplier(2);
console.log(double(7)); // 14
```

5.2.1 Summary

- Functions can return other functions, allowing **dynamic creation of customized utilities**.
- This pattern supports **function factories**, **currying**, and other composable designs.
- Returning functions enhances code modularity, flexibility, and reuse.

In the next section, we'll explore practical patterns like **function transformers** and **wrappers** that build on these ideas to modify and enhance function behavior.

5.3 Practical Patterns: Function Transformers and Wrappers

Higher-order functions not only let you pass and return functions—they also enable **transforming and enhancing existing functions** without changing their original code. This leads to **modular**, **reusable**, and **DRY (Don't Repeat Yourself)** patterns like **decorators**, **wrappers**, and **memoizers**.

What Are Function Wrappers and Transformers?

A **function wrapper** is a higher-order function that takes an existing function and returns a new function that **adds extra behavior** before, after, or around the original one.

This allows you to:

- Add logging or debugging,
- Control execution rate (throttling/debouncing),
- Cache expensive results (memoization),
- Handle errors or retries.

All without modifying the original function's implementation.

Example 1: Logger Wrapper

A logger wrapper prints the arguments and the result whenever the wrapped function is called.

```
function loggerWrapper(fn) {
  return function (...args) {
    console.log('Calling function with arguments:', args);
    const result = fn(...args);
    console.log('Function returned:', result);
    return result;
  };
}

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

// Wrap the add function
const loggedAdd = loggerWrapper(add);

loggedAdd(3, 4);
// Output:
// Calling function with arguments: [3, 4]
// Function returned: 7
```

The wrapper does not change `add` itself; it simply adds logging behavior around it.

Example 2: Memoization (Caching Results)

Memoization is a common pattern that caches function results to avoid repeating expensive computations.

```
function memoize(fn) {
  const cache = new Map();
  return function (...args) {
    const key = JSON.stringify(args);
    if (cache.has(key)) {
      console.log('Fetching from cache for args:', args);
      return cache.get(key);
    }
    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
}

// Example: Slow factorial function
function slowFactorial(n) {
  if (n === 0) return 1;
  return n * slowFactorial(n - 1);
}

// Create memoized version
const fastFactorial = memoize(slowFactorial);

console.log(fastFactorial(5)); // Computes and caches
console.log(fastFactorial(5)); // Retrieves from cache
```

Memoization improves performance and keeps your code clean by separating caching logic

from the original function.

Why Use These Patterns?

- **Modularity:** You keep core functions simple and focused on their main task.
- **Reusability:** Wrappers can be applied to many functions without rewriting code.
- **Separation of concerns:** Logging, caching, or throttling logic is isolated and reusable.
- **Maintainability:** Adding or removing wrappers is easy, improving code flexibility.

5.3.1 Summary

- Function wrappers and transformers are higher-order functions that **enhance existing functions without modifying them**.
- Common patterns include **logging wrappers** that print inputs and outputs, and **memoization** that caches results for efficiency.
- These techniques promote modular, DRY, and maintainable code—key goals in functional programming.

With these tools, you can elegantly extend functionality while keeping your core functions clean and pure.

Chapter 6.

Closures and Scope

1. Understanding Closures
2. Lexical Scope in Functional Code
3. Practical Uses: Memoization, Encapsulation

6 Closures and Scope

6.1 Understanding Closures

Closures are a fundamental and powerful concept in JavaScript that enable functions to **"remember"** the environment in which they were created. This ability lets functions access variables from their outer (enclosing) scopes even after those outer functions have finished executing.

What Is a Closure?

A **closure** is created when an inner function **retains access** to variables from its outer function, preserving that scope for later use.

In other words, a closure is a function bundled together with references to its surrounding state.

Step-by-Step Example of a Closure

```
function outer() {
  const message = 'Hello from outer!';

  function inner() {
    console.log(message);
  }

  return inner;  // Return the inner function
}

const closureFunction = outer(); // outer() runs and returns inner
closureFunction();               // Logs: "Hello from outer!"
```

What's happening here?

1. When `outer()` is called, it declares a variable `message` and defines an inner function `inner()` that logs `message`.
2. `outer()` returns the `inner` function itself, **not** the result of calling `inner`.
3. Even though `outer()` finishes execution, the returned function `closureFunction` still **remembers** the variable `message` from `outer`'s scope.
4. Calling `closureFunction()` logs `"Hello from outer!"` because of this retained environment — this is the closure in action.

Closure vs. Simple Function Scope

- **Function scope** means variables declared inside a function are accessible only within that function.

  `js try function sayHi() {  const greeting = 'Hi!';  console.log(greeting); // Works here } console.log(greeting); // Error: greeting is not defined`

- **Closure** allows an inner function to access the outer function's scoped variables **even after** the outer function has completed.

Without closures, once `outer()` finishes, its local variables like `message` would be lost. But closures keep them alive as long as the inner function exists.

Why Are Closures Useful?

- **Data encapsulation:** You can create private variables that only certain functions can access.
- **Stateful functions:** Functions can maintain state across calls without using global variables.
- **Function factories:** Closures enable creating customized functions dynamically.

6.1.1 Summary

- A **closure** is a function that retains access to variables from its outer scope, even after the outer function returns.
- This allows inner functions to "remember" the environment where they were created.
- Closures differ from simple function scope by preserving variables beyond the lifetime of the outer function call.
- They are a powerful tool for encapsulation, maintaining state, and writing flexible functional code.

Next, we will explore how **lexical scope** underpins closures and affects variable resolution in your JavaScript programs.

6.2 Lexical Scope in Functional Code

Understanding **lexical scope** is key to mastering closures and writing effective functional JavaScript. Lexical scoping defines **how variables are resolved** in nested functions based on where those functions are **defined**, not where they are **called**.

What Is Lexical Scope?

Lexical scope means that the accessibility of variables is determined by their physical placement in the source code—i.e., the structure of nested functions and blocks at the time the code is written.

In JavaScript, **functions form scopes**, and each scope has access to variables declared in its own scope **and** all outer scopes **above** it in the scope chain.

Variables Are Resolved at Definition Time

Functions **capture variables from the scopes they are defined in**, regardless of where or when they are later called.

Example: Lexical Scoping and Closures

```
function outer() {
  const outerVar = 'I am from outer';

  function inner() {
    console.log(outerVar);
  }

  return inner;
}

const fn = outer();
fn(); // Logs: "I am from outer"
```

- The inner function `inner` remembers the `outerVar` from its **lexical environment** — where it was defined.
- Even though `fn` is called outside of `outer()`, it still accesses `outerVar` from the scope chain.

Shadowing: Inner Variables Hide Outer Variables

If an inner scope declares a variable with the same name as an outer scope, the inner variable **shadows** (overrides) the outer one within its scope.

```
function outer() {
  const value = 'outer value';

  function inner() {
    const value = 'inner value'; // Shadows outer 'value'
    console.log(value);
  }

  inner();
}

outer(); // Logs: "inner value"
```

Variable Lookup: How JavaScript Resolves Identifiers

When a variable is referenced, JavaScript looks for it:

1. In the **current scope**.
2. If not found, it looks in the **next outer scope**, moving outward through the scope chain.
3. Continues until it reaches the **global scope**.
4. If not found anywhere, it results in a **ReferenceError**.

Example:

```
const globalVar = 'global';

function outer() {
  const outerVar = 'outer';

  function inner() {
    console.log(outerVar);  // Found in outer scope
    console.log(globalVar); // Found in global scope
  }

  inner();
}

outer();
// Logs:
// "outer"
// "global"
```

Why Lexical Scope Matters in Functional Programming

- **Predictability:** Since variables are resolved by their position in code, behavior is consistent and easy to reason about.
- **Closures:** Lexical scoping enables closures to reliably capture variables from outer environments.
- **Encapsulation:** Variables in inner scopes can safely shadow outer variables without side effects.

6.2.1 Summary

- **Lexical scope** means variable accessibility is based on the function's **location in the code**, not where it's called.
- Functions carry a **scope chain** of all their enclosing scopes, which they use to resolve variables.
- Variables in inner scopes can **shadow** outer ones, creating predictable lookup rules.
- This scoping model is essential for closures and the modular, maintainable code style of functional programming.

Next, we will see how closures and lexical scope combine in practical patterns like memoization and encapsulation.

6.3 Practical Uses: Memoization, Encapsulation

Closures are not just an abstract concept—they're incredibly useful tools that enable powerful functional programming techniques like **memoization** and **encapsulation**. These patterns help write efficient, clean, and maintainable code by leveraging the ability of functions to "remember" their environment.

Memoization: Caching Expensive Function Results

Memoization is an optimization technique that caches the results of expensive function calls and returns the cached result when the same inputs occur again. Closures make it easy to store this cache privately within the function.

Example: Memoized Fibonacci Function The classic Fibonacci sequence calculation can be inefficient due to repeated calculations. Let's memoize it using closures:

```
function memoizedFibonacci() {
  const cache = {}; // Private cache stored in closure

  function fib(n) {
    if (n <= 1) return n;

    // Check cache
    if (cache[n]) {
      console.log(`Fetching fib(${n}) from cache`);
      return cache[n];
    }

    // Compute and cache result
    const result = fib(n - 1) + fib(n - 2);
    cache[n] = result;
    return result;
  }

  return fib;
}

const fib = memoizedFibonacci();

console.log(fib(10)); // Computes and caches intermediate results
console.log(fib(10)); // Fetches result directly from cache
```

How it works:

- `memoizedFibonacci` returns the `fib` function which has access to the private `cache` object.
- Every call to `fib` uses the cache to avoid recomputing known values.
- The cache is **encapsulated** and cannot be accessed or mutated externally.

Encapsulation: Hiding Private State

Closures allow you to **hide internal variables** and expose only the functions that operate on them, preventing accidental or unauthorized modifications.

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

  return {
    increment() {
      count += 1;
      return count;
    },
    decrement() {
      count -= 1;
      return count;
    },
    getCount() {
      return count;
    }
  };
}

const counter = createCounter();

console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.getCount());  // 2
console.log(counter.decrement()); // 1

// Direct access to count is impossible:
// console.log(counter.count); // undefined
```

Example: Counter with Private State Why use closures here?

- The `count` variable is private—only accessible inside `createCounter`.
- External code cannot directly mutate `count`, ensuring controlled access through the API.
- This encapsulation protects state integrity and reduces bugs.

6.3.1 Summary

- **Memoization** uses closures to **cache results** privately inside functions, improving performance by avoiding redundant calculations.
- **Encapsulation** leverages closures to **hide internal state**, exposing only controlled interfaces to manipulate data safely.
- These patterns show how closures enable writing efficient, modular, and maintainable functional code in JavaScript.

In the next chapter, we'll explore how these principles combine with asynchronous programming for even more powerful abstractions.

Chapter 7.

Function Composition and Pipelining

1. Function Chaining vs Nesting
2. `compose()` and `pipe()` Utilities
3. Building Reusable Data Pipelines

7 Function Composition and Pipelining

7.1 Function Chaining vs Nesting

When combining multiple functions to transform data or perform a sequence of operations, two common approaches emerge: **function nesting** and **function chaining**. Both are ways to compose functions, but they differ in syntax, readability, and maintainability.

Function Nesting

Function nesting means calling functions **inside one another**, where the output of the innermost function becomes the input of the next outer function.

The syntax looks like this:

```
const result = f(g(h(x)));
```

Here, `h` is called first with `x`, then `g` receives `h(x)`, and finally `f` processes `g(h(x))`.

```
function double(n) {
  return n * 2;
}

function increment(n) {
  return n + 1;
}

function square(n) {
  return n * n;
}

const x = 3;
const result = double(increment(square(x))); // double(increment(9)) => double(10) => 20

console.log(result); // 20
```

Example: Nesting to transform a number

Function Chaining

Function chaining executes functions **sequentially** on data, typically by using intermediate variables or methods that return objects supporting chaining.

Chaining looks like this:

```
const result = x.method1().method2().method3();
```

This style is common with arrays and libraries like Lodash or jQuery.

```
const numbers = [1, 2, 3, 4];

const result = numbers
  .map(n => n * n)       // square each number
  .map(n => n + 1)       // increment each number
  .map(n => n * 2);      // double each number

console.log(result); // [4, 10, 20, 32]
```

Example: Chaining with Array Methods Each method returns a new array, allowing seamless chaining of transformations.

Readability and Maintainability: Pros and Cons

Aspect	Function Nesting	Function Chaining
Readability	Can become hard to read when deeply nested(right-to-left reading)	Reads left-to-right, following the flow of data transformations
Debugging	Difficult to insert debug statementsat intermediate steps	Easy to add logs or breakpoints after any step
Maintainability	Nested calls can be confusingto extend or modify	Clear step-by-step transformations make updates straightforward
Performance	Similar performance, though chaining often creates intermediate objects	May create intermediate objects but more readable code is preferred

Making Chaining More Declarative

Chaining encourages a **declarative style**, where you describe *what* to do in sequence, rather than *how* to nest function calls.

For example, utility functions like `compose` or `pipe` (covered later) allow you to write function composition in a clearer, linear way, improving maintainability without sacrificing expressiveness.

7.1.1 Summary

- **Function nesting** involves calling functions inside each other and reads from the innermost to outermost function.
- **Function chaining** involves calling methods or functions sequentially, reading top to bottom in a linear flow.
- Chaining tends to be more readable and easier to maintain, especially when working with sequences of data transformations.
- Thinking about chaining as a declarative pipeline of operations can help write clearer, more maintainable code.

In the next section, we will explore utility functions like `compose()` and `pipe()` that help formalize and simplify function composition patterns.

7.2 `compose()` and `pipe()` Utilities

When working with multiple functions, writing nested calls like `f(g(h(x)))` can get confusing. To simplify this, functional programming often uses utilities like **`compose()`** and **`pipe()`** to create new functions by combining several smaller ones into one.

What Are `compose()` and `pipe()`?

Both `compose` and `pipe` **combine multiple functions into a single function** that executes them in sequence. The key difference lies in the order of execution:

- **`compose(f, g, h)`** calls the functions **right-to-left**: `compose(f, g, h)(x)` is equivalent to `f(g(h(x)))`.
- **`pipe(f, g, h)`** calls the functions **left-to-right**: `pipe(f, g, h)(x)` is equivalent to `h(g(f(x)))`.

Minimal Implementation of `compose()`

```
const compose = (...fns) => input =>
  fns.reduceRight((acc, fn) => fn(acc), input);
```

- `reduceRight` starts from the rightmost function and applies each function to the accumulated value.

Minimal Implementation of `pipe()`

```
const pipe = (...fns) => input =>
  fns.reduce((acc, fn) => fn(acc), input);
```

- `reduce` processes functions from left to right.

Practical Example: String Transformations

Let's combine simple string functions like `trim()`, `toLowerCase()`, and `replace()`.

```
const trim = str => str.trim();
const toLowerCase = str => str.toLowerCase();
const removeSpaces = str => str.replace(/\s+/g, '');

const input = "  Hello Functional Programming  ";
```

```
const cleanString = compose(removeSpaces, toLowerCase, trim);

console.log(cleanString(input));
// Output: "hellofunctionalprogramming"
```

Using `compose()` (right-to-left) Execution flow: `trim(input)` → `toLowerCase(result)` → `removeSpaces(result)`

```
const cleanStringPipe = pipe(trim, toLowerCase, removeSpaces);

console.log(cleanStringPipe(input));
// Output: "hellofunctionalprogramming"
```

Using `pipe()` (left-to-right) Execution flow: `trim(input)` → `toLowerCase(result)` → `removeSpaces(result)`

Full runnable code:

```
// Minimal Implementation of compose (right-to-left)
const compose = (...fns) => input =>
  fns.reduceRight((acc, fn) => fn(acc), input);

// Minimal Implementation of pipe (left-to-right)
const pipe = (...fns) => input =>
  fns.reduce((acc, fn) => fn(acc), input);

// String transformation functions
const trim = str => str.trim();
const toLowerCase = str => str.toLowerCase();
const removeSpaces = str => str.replace(/\s+/g, '');

const input = "  Hello Functional Programming  ";

// Using compose
const cleanString = compose(removeSpaces, toLowerCase, trim);
console.log("Using compose:", cleanString(input));
// Output: hellofunctionalprogramming

// Using pipe
const cleanStringPipe = pipe(trim, toLowerCase, removeSpaces);
console.log("Using pipe:", cleanStringPipe(input));
// Output: hellofunctionalprogramming
```

When to Use `compose()` vs. `pipe()`

- Use **`compose()`** if you prefer reading function calls from the inside out, mirroring mathematical composition notation.
- Use **`pipe()`** if you need a **more natural left-to-right reading order** that matches the order of data transformations.

7.2.1 Summary

- `compose()` and `pipe()` create new functions by combining smaller ones, making complex transformations easier to read and maintain.
- `compose()` executes functions right-to-left, while `pipe()` executes left-to-right.
- Both enable you to build clear, declarative pipelines from simple reusable functions.

In the next section, we'll explore how to use these tools to build reusable, flexible data pipelines for real-world scenarios.

7.3 Building Reusable Data Pipelines

One of the most powerful applications of function composition and pipelining is building **reusable, declarative data transformation pipelines**. These pipelines process data step-by-step by chaining pure functions, making your code modular, maintainable, and easy to understand.

Why Use Data Pipelines?

- **Modularity:** Each function in the pipeline focuses on a single, clear task.
- **Separation of Concerns:** Data transformations are decoupled, so you can update or reuse parts independently.
- **Readability:** The pipeline reads like a sequence of operations, describing what happens to the data clearly.
- **Testability:** Pure functions are easier to test individually.

Example: Processing an Array of User Objects

Let's say we have a list of users, and we want to:

1. Filter out inactive users.
2. Extract just their email addresses.
3. Sort the emails alphabetically.

```
const users = [
  { id: 1, name: 'Alice', email: 'alice@example.com', active: true },
  { id: 2, name: 'Bob', email: 'bob@example.com', active: false },
  { id: 3, name: 'Charlie', email: 'charlie@example.com', active: true }
];
```

Sample Data

```
// Filter active users
const filterActiveUsers = users =>
```

```
  users.filter(user => user.active);

// Extract emails
const extractEmails = users =>
  users.map(user => user.email);

// Sort emails alphabetically
const sortEmails = emails =>
  [...emails].sort(); // Spread to avoid mutating original array
```

Step 1: Create Small, Pure Transformation Functions

```
const pipe = (...fns) => input =>
  fns.reduce((acc, fn) => fn(acc), input);

const getActiveSortedEmails = pipe(
  filterActiveUsers,
  extractEmails,
  sortEmails
);

console.log(getActiveSortedEmails(users));
// Output: ['alice@example.com', 'charlie@example.com']
```

Step 2: Compose a Pipeline Using `pipe()`

Full runnable code:

```
// Sample Data
const users = [
  { id: 1, name: 'Alice', email: 'alice@example.com', active: true },
  { id: 2, name: 'Bob', email: 'bob@example.com', active: false },
  { id: 3, name: 'Charlie', email: 'charlie@example.com', active: true }
];

// Small, pure transformation functions

// Filter active users
const filterActiveUsers = users =>
  users.filter(user => user.active);

// Extract emails
const extractEmails = users =>
  users.map(user => user.email);

// Sort emails alphabetically
const sortEmails = emails =>
  [...emails].sort(); // avoid mutating original array

// Compose pipeline
const pipe = (...fns) => input =>
  fns.reduce((acc, fn) => fn(acc), input);

// Define the full transformation
const getActiveSortedEmails = pipe(
  filterActiveUsers,
```

```
  extractEmails,
  sortEmails
);

// Run and output result
console.log(getActiveSortedEmails(users));
// Output: ['alice@example.com', 'charlie@example.com']
```

How This Pipeline Works

- The `users` array is passed into `getActiveSortedEmails`.
- It flows through each function in order:
  - `filterActiveUsers` returns only active users.
  - `extractEmails` maps users to their emails.
  - `sortEmails` returns a sorted copy of the emails.
- Each function is pure and focused, making the pipeline clear and testable.

Best Practices for Pipeline Components

- **Name functions clearly:** Names like `filterActiveUsers` or `extractEmails` communicate intent and improve readability.
- **Keep functions small and focused:** Each should perform one transformation or concern.
- **Avoid side effects:** Functions should not mutate inputs; instead, return new transformed data.
- **Use immutable patterns:** Spread operators or methods like `map`, `filter`, and `sort` (on copies) help keep data immutable.
- **Document assumptions:** Note expected input/output shapes when necessary for clarity.

7.3.1 Summary

- Building data pipelines with function composition creates clean, modular, and reusable code.
- Pipelines promote clear separation of concerns by breaking transformations into pure, focused functions.
- Using utilities like `pipe()` makes pipelines declarative and easy to read.
- Thoughtful naming and small function design improve maintainability and testing.

In the next chapter, we'll explore how these functional patterns extend to asynchronous programming for handling complex workflows.

Chapter 8.

Working with Arrays Functionally

1. `map()`, `filter()`, `reduce()`
2. Chaining Array Methods
3. Practical Example: Data Transformation Pipeline

8 Working with Arrays Functionally

8.1 `map()`, `filter()`, `reduce()`

JavaScript's array methods `map()`, `filter()`, and `reduce()` are the cornerstone of functional programming with arrays. They allow you to **transform**, **select**, and **aggregate** data declaratively and immutably.

`map()`: Transforming Data

The `map()` method creates a **new array** by applying a function to each element of the original array. Use it when you want to **transform** each item without changing the original array.

```
const numbers = [1, 2, 3, 4];

// Double each number
const doubled = numbers.map(n => n * 2);

console.log(doubled); // [2, 4, 6, 8]
```

`filter()`: Selecting Data

The `filter()` method creates a **new array** containing only the elements that satisfy a condition defined by a predicate function. Use it to **select** or **exclude** elements.

```
const numbers = [1, 2, 3, 4, 5];

// Select only even numbers
const evens = numbers.filter(n => n % 2 === 0);

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

`reduce()`: Aggregating or Accumulating Data

The `reduce()` method **reduces** the array to a single value by repeatedly applying a reducer function that accumulates results.

It's useful for:

- Summing values
- Combining objects
- Flattening arrays
- And many other aggregations

```
const numbers = [1, 2, 3, 4];

// Sum all numbers
const sum = numbers.reduce((accumulator, current) => accumulator + current, 0);

console.log(sum); // 10
```

Example: Using `map()`, `filter()`, and `reduce()` Together

Suppose you have an array of user objects:

```
const users = [
  { name: 'Alice', age: 25, active: true },
  { name: 'Bob', age: 30, active: false },
  { name: 'Carol', age: 35, active: true }
];
```

- Use `filter()` to get active users.
- Use `map()` to extract their ages.
- Use `reduce()` to calculate the total age.

```
const users = [
  { name: 'Alice', age: 25, active: true },
  { name: 'Bob', age: 30, active: false },
  { name: 'Carol', age: 35, active: true }
];
const totalActiveAge = users
  .filter(user => user.active)
  .map(user => user.age)
  .reduce((sum, age) => sum + age, 0);

console.log(totalActiveAge); // 60 (25 + 35)
```

8.1.1 Summary

- `map()` transforms each element into a new form.
- `filter()` selects elements based on a condition.
- `reduce()` combines elements to produce a single value.

Together, these methods allow expressive, concise, and immutable data manipulation in JavaScript functional programming.

8.2 Chaining Array Methods

One of the most powerful features of JavaScript's array methods is that they can be **chained together** to create expressive and readable data transformation pipelines. Chaining allows you to perform multiple operations on data step-by-step, without mutating the original array or creating unnecessary intermediate variables.

Why Chain Array Methods?

- **Expressiveness:** You can describe a sequence of transformations clearly and succinctly.
- **Immutability:** Each method returns a new array or value, leaving the original data untouched.
- **No intermediate variables:** Chaining reduces clutter by eliminating temporary variables holding intermediate results.
- **Maintainability:** The flow of data transformations is linear and easier to follow.

Example: Chaining with Products Array

Suppose you have an array of product objects and want to:

1. Filter to include only available products.
2. Extract their prices.
3. Calculate the total price.

```
const products = [
  { name: 'Laptop', price: 1000, available: true },
  { name: 'Phone', price: 500, available: false },
  { name: 'Tablet', price: 750, available: true }
];

const totalAvailablePrice = products
  .filter(product => product.available)       // Step 1: Filter available products
  .map(product => product.price)              // Step 2: Map to prices
  .reduce((total, price) => total + price, 0); // Step 3: Sum the prices

console.log(totalAvailablePrice); // 1750 (1000 + 750)
```

How Chaining Works Here

- `filter()` returns a new array with only available products.
- `map()` transforms that array into an array of prices.
- `reduce()` aggregates those prices into a total sum.

All without modifying the original `products` array.

Balancing Performance and Readability

- **Performance:** Each array method creates a new intermediate array, which could add overhead in very large data sets.
- **Optimization:** Sometimes, it's worth combining transformations manually or using libraries with lazy evaluation (e.g., RxJS, Lodash/fp).
- **Readability:** Prefer clear, maintainable code. Chaining often improves readability by expressing intent clearly.
- **Profiling:** If performance becomes a concern, profile your code before optimizing prematurely.

8.2.1 Summary

- Chaining array methods like `filter()`, `map()`, and `reduce()` lets you build clear, readable data transformation pipelines.
- This pattern promotes immutability by creating new arrays and avoids clutter with intermediate variables.
- While chaining is usually efficient and expressive, be mindful of performance in large-scale data processing.
- Aim for a balance between readability and performance based on your specific use case.

Next, we'll see how to combine these techniques into practical, reusable data transformation pipelines.

8.3 Practical Example: Data Transformation Pipeline

To see how `map()`, `filter()`, and `reduce()` work together in a real-world scenario, let's build a data transformation pipeline that calculates **total sales for items priced above a certain threshold**. This example demonstrates the power of chaining pure functions to solve common problems cleanly and declaratively.

Problem Statement

Given an array of sales items, we want to:

- Filter out items priced below a specified threshold.
- Extract the total price of each qualifying item (price × quantity).
- Calculate the overall total sales amount for those items.

Sample Data

```
const sales = [
  { item: 'Laptop', price: 1000, quantity: 4 },
  { item: 'Phone', price: 500, quantity: 10 },
  { item: 'Tablet', price: 750, quantity: 5 },
  { item: 'Headphones', price: 100, quantity: 20 }
];
```

Step 1: Break Into Reusable Functions

Let's create small pure functions for each step:

```
// Filter items above price threshold
const filterExpensiveItems = (threshold) => (items) =>
  items.filter(item => item.price > threshold);

// Calculate total price per item
const calculateItemTotal = items =>
```

```
  items.map(item => ({ ...item, total: item.price * item.quantity }));

// Sum all total prices
const sumTotalSales = items =>
  items.reduce((acc, item) => acc + item.total, 0);
```

Step 2: Build the Pipeline with Chaining

```
const priceThreshold = 300;

const totalSales = sumTotalSales(
  calculateItemTotal(
    filterExpensiveItems(priceThreshold)(sales)
  )
);

console.log(totalSales);
// Output: 1000*4 + 500*10 + 750*5 = 4000 + 5000 + 3750 = 12,750
```

Step 3: Refactor Using a `pipe` Utility for Readability

Let's use the `pipe` function introduced earlier to make the flow clearer:

```
const pipe = (...fns) => input =>
  fns.reduce((acc, fn) => fn(acc), input);

const calculateTotalSalesAboveThreshold = (threshold, items) =>
  pipe(
    filterExpensiveItems(threshold),
    calculateItemTotal,
    sumTotalSales
  )(items);

console.log(calculateTotalSalesAboveThreshold(priceThreshold, sales));
// 12750
```

Full runnable code:

```
// Sample Data
const sales = [
  { item: 'Laptop', price: 1000, quantity: 4 },
  { item: 'Phone', price: 500, quantity: 10 },
  { item: 'Tablet', price: 750, quantity: 5 },
  { item: 'Headphones', price: 100, quantity: 20 }
];

// Step 1: Reusable pure functions

// Filter items above price threshold
const filterExpensiveItems = (threshold) => (items) =>
  items.filter(item => item.price > threshold);

// Calculate total price per item
const calculateItemTotal = items =>
```

```
  items.map(item => ({ ...item, total: item.price * item.quantity }));

// Sum all total prices
const sumTotalSales = items =>
  items.reduce((acc, item) => acc + item.total, 0);

// Step 2: Pipe utility
const pipe = (...fns) => input =>
  fns.reduce((acc, fn) => fn(acc), input);

// Step 3: Build pipeline and execute
const priceThreshold = 300;

const calculateTotalSalesAboveThreshold = (threshold, items) =>
  pipe(
    filterExpensiveItems(threshold),
    calculateItemTotal,
    sumTotalSales
  )(items);

// Run and print result
console.log(calculateTotalSalesAboveThreshold(priceThreshold, sales));
// Expected output: 12750
```

8.3.1 Explanation

- `filterExpensiveItems(threshold)` returns a function that filters the sales array for items with price > threshold.
- `calculateItemTotal` adds a `total` property for each item (`price * quantity`).
- `sumTotalSales` reduces the array to the sum of all item totals.
- Using `pipe`, we compose these transformations into a clear, reusable pipeline.

8.3.2 Summary

- Combining `filter()`, `map()`, and `reduce()` enables you to build complex data transformations declaratively.
- Breaking the pipeline into small, reusable pure functions improves clarity and testability.
- Using composition utilities like `pipe()` makes your data flow easier to read and maintain.

Try applying this approach to other real-world data processing problems, and watch your functional programming skills grow!

Chapter 9.

Functional String and Object Handling

1. Functional Techniques for Strings
2. Object Transformation with `Object.entries()`, `Object.fromEntries()`
3. Practical Example: Parsing and Reshaping JSON

9 Functional String and Object Handling

9.1 Functional Techniques for Strings

Strings in JavaScript are **immutable**, meaning that any operation on a string returns a **new string** rather than modifying the original. This immutability fits perfectly with functional programming principles, allowing you to apply **pure functions** and chain string transformations safely without side effects.

Chaining String Methods

Just like array methods, many string methods can be chained together to create **clear, declarative** transformations. Some common methods used in functional string handling include:

- `trim()` — removes whitespace from both ends of a string.
- `toLowerCase()` / `toUpperCase()` — converts string case.
- `split()` — splits a string into an array based on a delimiter.
- `replace()` — replaces parts of a string using substrings or regular expressions.

Example 1: Normalizing User Input

User input often contains unwanted spaces, inconsistent capitalization, or other formatting issues. Here's how you can clean and normalize a username:

```
const normalizeUsername = username =>
  username
    .trim()                    // Remove leading/trailing whitespace
    .toLowerCase()             // Convert all characters to lowercase
    .replace(/\s+/g, '_');     // Replace spaces with underscores

console.log(normalizeUsername('  John Doe  '));
// Output: "john_doe"
```

Example 2: Formatting a Sentence

You might want to format text by trimming spaces, fixing capitalization, and removing unwanted characters:

```
const formatSentence = sentence =>
  sentence
    .trim()                           // Remove extra whitespace
    .replace(/[^\w\s.]/g, '')         // Remove special characters except word chars, spaces, and period
    .toLowerCase()                    // Convert to lowercase
    .replace(/(^\w|\.\s\w)/g, c => c.toUpperCase()); // Capitalize first letters

const input = '   hello, world! this is functional programming.   ';
console.log(formatSentence(input));
// Output: "Hello world. This is functional programming."
```

Immutability and Avoiding Side Effects

Because strings are immutable in JavaScript:

- String methods **do not change** the original string but **return new strings**.
- This behavior helps keep functions **pure** and free from side effects.
- Avoid directly mutating strings or relying on global variables when performing transformations.

9.1.1 Summary

- Strings in JavaScript are immutable, which aligns well with functional programming.
- Chaining string methods like `trim()`, `toLowerCase()`, `split()`, and `replace()` enables clean, declarative transformations.
- Functional string handling avoids side effects, making your code more predictable and testable.
- Use small, focused functions to normalize or format text for better readability and reuse.

Next, we'll explore how to apply similar functional techniques when working with JavaScript objects.

9.2 Object Transformation with `Object.entries()`, `Object.fromEntries()`

JavaScript objects are often used to represent structured data, but unlike arrays, they don't have built-in methods like `map` or `filter`. Fortunately, the methods `Object.entries()` and `Object.fromEntries()` enable you to **transform objects functionally** by converting them to and from arrays of key-value pairs.

How They Work

- **`Object.entries(obj)`** converts an object into an array of `[key, value]` pairs.
- **`Object.fromEntries(array)`** converts an array of `[key, value]` pairs back into an object.

This opens the door to using array methods like `map()`, `filter()`, and `reduce()` to perform transformations on objects **without mutating the original**.

Example 1: Filtering Object Properties

Suppose you want to create a new object containing only certain keys.

```
const user = {
  name: 'Alice',
```

```
  age: 30,
  active: true,
  role: 'admin'
};

// Filter out only keys with truthy values
const filteredUser = Object.fromEntries(
  Object.entries(user)
    .filter(([key, value]) => Boolean(value))
);

console.log(JSON.stringify(filteredUser,null,2));
// Output: { name: 'Alice', age: 30, active: true, role: 'admin' }
```

If you need to filter out a specific key, for example `'role'`:

```
const user = {
  name: 'Alice',
  age: 30,
  active: true,
  role: 'admin'
};

const withoutRole = Object.fromEntries(
  Object.entries(user)
    .filter(([key]) => key !== 'role')
);

console.log(JSON.stringify(withoutRole,null,2));
// Output: { name: 'Alice', age: 30, active: true }
```

Example 2: Renaming Object Keys

You can transform keys by mapping over entries and returning new keys.

```
const user = {
  name: 'Alice',
  age: 30,
  active: true,
  role: 'admin'
};

const renameKeys = obj =>
  Object.fromEntries(
    Object.entries(obj).map(([key, value]) => {
      const newKey = key === 'name' ? 'fullName' : key;
      return [newKey, value];
    })
  );

console.log(JSON.stringify(renameKeys(user),null,2));
// Output: { fullName: 'Alice', age: 30, active: true, role: 'admin' }
```

Example 3: Modifying Values

You can also change values by mapping over the entries.

```
const user = {
  name: 'Alice',
  age: 30,
  active: true,
  role: 'admin'
};

const capitalizeStrings = obj =>
  Object.fromEntries(
    Object.entries(obj).map(([key, value]) => {
      if (typeof value === 'string') {
        return [key, value.toUpperCase()];
      }
      return [key, value];
    })
  );

console.log(JSON.stringify(capitalizeStrings(user),null,2));
// Output: { name: 'ALICE', age: 30, active: true, role: 'ADMIN' }
```

Emphasizing Immutability

- Each of these transformations returns a **new object**.
- The original object remains **unchanged**, promoting functional purity.
- Avoid direct mutations like `obj[key] = newValue` inside these transformations.

9.2.1 Summary

- `Object.entries()` and `Object.fromEntries()` let you convert objects to arrays and back, enabling powerful functional transformations.
- You can filter, rename, or modify object properties using array methods like `filter()` and `map()`.
- These techniques keep your code **immutable** and **declarative**, aligning well with functional programming principles.

In the next section, we'll apply these concepts to parse and reshape JSON data in practical scenarios.

9.3 Practical Example: Parsing and Reshaping JSON

Working with JSON data is a common task in JavaScript applications, especially when consuming APIs. Functional programming techniques allow you to parse, transform, and

reshape JSON data cleanly and predictably.

Scenario

Imagine you receive the following JSON response representing a list of blog posts, each with nested author info:

```
{
  "posts": [
    {
      "id": 1,
      "title": "Functional Programming in JS",
      "content": "An introduction...",
      "author": { "id": 101, "name": "Alice", "active": true },
      "tags": ["javascript", "functional"]
    },
    {
      "id": 2,
      "title": "Object Manipulation",
      "content": "Working with objects...",
      "author": { "id": 102, "name": "Bob", "active": false },
      "tags": ["javascript", "objects"]
    }
  ]
}
```

Your goal is to:

- Extract only **active authors' posts**.
- Restructure each post to include only the `id`, `title`, and `authorName`.
- Return a new JSON object with this filtered and reshaped data.

Step 1: Parse the JSON

```
const jsonData = `{
  "posts": [
    {
      "id": 1,
      "title": "Functional Programming in JS",
      "content": "An introduction...",
      "author": { "id": 101, "name": "Alice", "active": true },
      "tags": ["javascript", "functional"]
    },
    {
      "id": 2,
      "title": "Object Manipulation",
      "content": "Working with objects...",
      "author": { "id": 102, "name": "Bob", "active": false },
      "tags": ["javascript", "objects"]
    }
  ]
}`;

const data = JSON.parse(jsonData);
```

Step 2: Define Pure Transformation Functions

```
// Filter posts with active authors
const filterActiveAuthors = posts =>
  posts.filter(post => post.author.active);

// Map posts to a new structure
const reshapePosts = posts =>
  posts.map(post => ({
    id: post.id,
    title: post.title,
    authorName: post.author.name
  }));
```

Step 3: Build the Transformation Pipeline

```
const transformData = data => {
  const activePosts = filterActiveAuthors(data.posts);
  const reshapedPosts = reshapePosts(activePosts);
  return { posts: reshapedPosts };
};

const transformed = transformData(data);

console.log(JSON.stringify(transformed, null, 2));
```

Output

```
{
  "posts": [
    {
      "id": 1,
      "title": "Functional Programming in JS",
      "authorName": "Alice"
    }
  ]
}
```

Full runnable code:

```
// Step 1: Parse the JSON
const jsonData = `{
  "posts": [
    {
      "id": 1,
      "title": "Functional Programming in JS",
      "content": "An introduction...",
      "author": { "id": 101, "name": "Alice", "active": true },
      "tags": ["javascript", "functional"]
    },
    {
      "id": 2,
      "title": "Object Manipulation",
```

```
      "content": "Working with objects...",
      "author": { "id": 102, "name": "Bob", "active": false },
      "tags": ["javascript", "objects"]
    }
  ]
}`;

const data = JSON.parse(jsonData);

// Step 2: Define Pure Transformation Functions

// Filter posts with active authors
const filterActiveAuthors = posts =>
  posts.filter(post => post.author.active);

// Map posts to a new structure
const reshapePosts = posts =>
  posts.map(post => ({
    id: post.id,
    title: post.title,
    authorName: post.author.name
  }));

// Step 3: Build the Transformation Pipeline
const transformData = data => {
  const activePosts = filterActiveAuthors(data.posts);
  const reshapedPosts = reshapePosts(activePosts);
  return { posts: reshapedPosts };
};

const transformed = transformData(data);

// Output the result
console.log(JSON.stringify(transformed, null, 2));
```

9.3.1 Why Functional?

- **Pure functions:** `filterActiveAuthors` and `reshapePosts` do not modify input but return new arrays.
- **Clear, testable transformations:** Each step has a single responsibility and can be tested independently.
- **Predictability:** The pipeline always produces the same output for the same input.
- **Immutability:** Original JSON data remains unchanged.

9.3.2 Summary

This example shows how to parse, filter, map, and rebuild JSON data using functional techniques. By breaking the problem into small, pure functions and composing them, you

gain readable, maintainable, and predictable code—key advantages of functional programming in JavaScript.

Chapter 10.

Recursion in Functional Programming

1. Why Functional Programming Uses Recursion
2. Writing Recursive Functions
3. Tail Call Optimization (and When It's Not Available)

10 Recursion in Functional Programming

10.1 Why Functional Programming Uses Recursion

In functional programming, **recursion** is a fundamental technique for expressing repetition and iteration without relying on mutable state or imperative loops. This makes recursion a natural fit for the functional paradigm.

The Absence of Mutable State and Loops

Traditional loops like `for` or `while` often rely on mutable variables that change with each iteration:

```
let sum = 0;
for (let i = 1; i <= 5; i++) {
  sum += i;
}
console.log(sum); // 15
```

However, **pure functional programming avoids mutable state** to ensure predictability and easier reasoning about code. Since loops inherently depend on updating state, they don't align well with pure FP principles.

Recursion as a Declarative Alternative

Recursion solves this problem by defining a function that calls itself with a **smaller or simpler input**, gradually reaching a base case without changing external state.

For example, calculating factorial (n!) recursively:

```
const factorial = n =>
  n === 0 ? 1 : n * factorial(n - 1);

console.log(factorial(5)); // 120
```

Here, instead of looping and mutating a variable, the function expresses the problem **declaratively**: the factorial of `n` is `n` times the factorial of `n-1`, until it reaches 0.

Working Naturally with Immutable Data

Recursion fits especially well with **immutable data structures**, since each recursive call works with its own inputs and returns new results without side effects.

Another example is the Fibonacci sequence:

```
const fibonacci = n =>
  n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2);

console.log(fibonacci(6)); // 8
```

This recursive definition directly models the mathematical formula without looping or mutable

variables.

10.1.1 Summary

- Functional programming avoids mutable state, making loops less suitable.
- **Recursion** replaces loops by expressing iterative processes declaratively.
- It naturally fits immutable data by working through self-contained function calls.
- Simple examples like factorial and Fibonacci highlight how recursion captures repetition functionally and elegantly.

10.2 Writing Recursive Functions

Recursion is a technique where a function calls itself to solve smaller instances of the same problem. Writing effective recursive functions in JavaScript requires understanding the **structure** of recursion and careful handling of termination conditions.

The Recursive Structure: Base Case and Recursive Case

Every recursive function has two essential parts:

1. **Base case:** The condition under which the recursion stops. It prevents infinite calls and typically handles the simplest input.
2. **Recursive case:** The part where the function calls itself with smaller or simpler arguments, gradually approaching the base case.

Example 1: Factorial Function

Let's revisit the factorial calculation to see this structure clearly:

```
const factorial = n => {
  if (n === 0) {           // Base case
    return 1;
  } else {                 // Recursive case
    return n * factorial(n - 1);
  }
};

console.log(factorial(5)); // 120
```

- **Base case:** When `n` is 0, return 1.
- **Recursive case:** Multiply `n` by factorial of `n-1`.

Example 2: Summing an Array Recursively

Recursion can also be used to process collections like arrays:

```
const sumArray = arr => {
  if (arr.length === 0) {   // Base case: empty array sum is 0
    return 0;
  } else {
    return arr[0] + sumArray(arr.slice(1)); // Recursive case: sum first element + sum of rest
  }
};

console.log(sumArray([1, 2, 3, 4, 5])); // 15
```

- **Base case:** Empty array returns 0.
- **Recursive case:** Sum the first element and recursively sum the rest of the array.

Example 3: Traversing Nested Data Structures

Recursion shines when dealing with deeply nested structures such as trees:

```
const countNodes = node => {
  if (!node.children || node.children.length === 0) {  // Base case: no children
    return 1;
  } else {
    return 1 + node.children.reduce((acc, child) => acc + countNodes(child), 0);
  }
};

const tree = {
  value: 1,
  children: [
    { value: 2, children: [] },
    {
      value: 3,
      children: [
        { value: 4, children: [] }
      ]
    }
  ]
};

console.log(countNodes(tree)); // 4
```

- **Base case:** Node with no children counts as 1.
- **Recursive case:** Count the current node plus all child nodes recursively.

Important Tips to Avoid Pitfalls

- Always ensure your **base case is reachable**; otherwise, recursion never stops.
- Beware of **stack overflow** errors if the recursion is too deep or base cases are missing.
- Consider iterative alternatives or techniques like **tail recursion** when working with large datasets (we will cover this next).
- Test your recursive functions with edge cases like empty inputs and minimum values.

10.2.1 Summary

- Recursive functions consist of a **base case** that stops recursion and a **recursive case** that breaks down the problem.
- Examples like factorial, summing arrays, and tree traversal illustrate common recursive patterns.
- Correctness is critical to avoid infinite recursion and stack overflow.
- Recursion offers elegant solutions for problems involving self-similar, nested, or iterative logic without mutable state.

10.3 Tail Call Optimization (and When It's Not Available)

Recursive functions are elegant but can run into a serious limitation: **stack overflow**. Each recursive call adds a new frame to the call stack, and if the recursion is too deep, your program can crash. **Tail Call Optimization (TCO)** is a technique that some programming languages and runtimes use to mitigate this problem by **reusing the current stack frame** for certain recursive calls.

What is Tail Call Optimization?

A **tail call** is a function call that is the **last operation** in a function before it returns. Tail call optimization allows the JavaScript engine to execute this call without adding a new stack frame, effectively turning recursion into iteration under the hood.

Example of a tail call:

```
function tailRecursiveFactorial(n, acc = 1) {
  if (n === 0) {
    return acc;
  }
  // The recursive call is the last thing executed (tail call)
  return tailRecursiveFactorial(n - 1, n * acc);
}

console.log(tailRecursiveFactorial(5)); // 120
```

In this function:

- The recursive call to `tailRecursiveFactorial` is the **last action**.
- The accumulated result `acc` carries the intermediate values.
- If TCO were supported, this would execute without growing the call stack.

Which Calls Are Tail Calls?

For a call to qualify as a tail call:

- It must be the **final action** before the function returns.
- No additional computation or operations should happen after the call.

For example:

```
function notTailRecursive(n) {
  return n * notTailRecursive(n - 1);  // NOT a tail call - multiplication happens after recursion
}
```

The multiplication happens **after** the recursive call returns, so this is **not** a tail call.

JavaScript Engine Support for TCO

While the ECMAScript 2015 (ES6) specification **requires** proper tail calls, **most current mainstream JavaScript engines (like V8 in Chrome and Node.js) do not fully implement TCO**.

This means that even tail-recursive functions may cause stack overflows in practice.

Alternatives When TCO is Not Available

Since TCO support is limited, here are strategies to avoid stack overflow:

1. **Use iterative loops** for deep or performance-critical recursion:

```
function iterativeFactorial(n) {
  let acc = 1;
  for (let i = n; i > 0; i--) {
    acc *= i;
  }
  return acc;
}
```

2. **Manual stack simulation:** Use a loop with a stack data structure to simulate recursion.
3. **Trampoline functions:** A more advanced technique to convert recursive calls into iteration by repeatedly invoking returned functions.

Practical Advice

- For small or moderate recursion depths, simple recursion is fine.
- For large inputs or deep recursion, prefer iterative or tail-recursive styles.
- Always test with edge cases to avoid stack overflow crashes.
- Stay updated on JavaScript engine developments for future TCO support.

10.3.1 Summary

- **Tail Call Optimization (TCO)** reuses stack frames for tail calls to improve recursion performance.
- Tail calls must be the last operation in a function for TCO to apply.
- JavaScript engines currently have inconsistent or limited TCO support.

- When TCO isn't available, use iterative solutions or other patterns to avoid stack overflow.
- Writing tail-recursive functions prepares your code for environments that support TCO in the future.

Next, you'll learn practical patterns and optimizations for working with recursion safely and effectively in JavaScript functional programming.

Chapter 11.

Currying and Partial Application

1. What Is Currying?
2. Currying vs Partial Application
3. Building Curried Utilities and Function Templates

11 Currying and Partial Application

11.1 What Is Currying?

Currying is a powerful functional programming technique that transforms a function with multiple arguments into a sequence of functions, each taking just one argument. Instead of calling a function with all arguments at once, you call a chain of functions, each receiving a single input.

Understanding Currying

Imagine a simple function that adds two numbers:

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

console.log(add(2, 3)); // 5
```

In a **curried** version, this function is transformed so you call it one argument at a time:

```
function curriedAdd(a) {
  return function (b) {
    return a + b;
  };
}

const addTwo = curriedAdd(2); // Returns a function waiting for the second argument
console.log(addTwo(3));       // 5

// Or call in one go by chaining:
console.log(curriedAdd(2)(3)); // 5
```

Here, `curriedAdd` takes one argument `a` and returns a new function that takes the second argument `b`. Only after both arguments have been provided is the sum computed.

Benefits of Currying

- **Partial application:** You can fix some arguments early and reuse the resulting functions with different remaining arguments. For example, `addTwo` always adds 2 to its input.
- **Function composition:** Curried functions are easier to compose, because they accept one argument at a time, fitting well into pipelines and higher-order functions.
- **Improved readability:** Breaking down functions into smaller, unary functions clarifies intent and enables modular design.

More General Example: Curried Multiply

```
const multiply = a => b => a * b;

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

const triple = multiply(3);
console.log(triple(5));  // 15

console.log(multiply(4)(6)); // 24
```

Here, `multiply` is a curried function that produces specialized multipliers like `double` and `triple` by partially applying the first argument.

11.1.1 Summary

- Currying transforms a multi-argument function into a chain of unary functions.
- It enables **partial application**, where you can fix some arguments early and reuse the rest.
- Currying promotes modularity, reuse, and better function composition patterns in functional programming.
- Calling curried functions looks like: `f(a)(b)(c)`, instead of `f(a, b, c)`.

In the next section, we'll explore how currying differs from partial application, and when to use each.

11.2 Currying vs Partial Application

Currying and partial application are closely related functional programming techniques that involve working with functions and their arguments—but they are **not the same**. Understanding their differences helps you choose the right tool for your code.

What Is Partial Application?

Partial application means taking a function with multiple arguments and **fixing some of those arguments** in advance, creating a new function that takes fewer arguments.

It does **not** necessarily transform the function into unary functions one by one, but rather pre-fills some arguments for convenience.

Currying Recap

- Transforms a function of multiple arguments into a **sequence of unary functions**.

- Each function takes exactly one argument.
- Allows calling a function one argument at a time: `f(a)(b)(c)`.

Partial Application Recap

- Fixes some arguments of a function upfront.
- Returns a new function with **fewer arguments**, but not necessarily unary.
- Arguments can be fixed in any order, not strictly left to right.

Practical Examples

Suppose we have a simple three-argument function:

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

Currying Example We can write a curried version:

```
const curriedVolume = length => width => height => length * width * height;

console.log(curriedVolume(2)(3)(4)); // 24

const volumeWithLength2 = curriedVolume(2); // returns width => height => ...
console.log(volumeWithLength2(3)(4));       // 24
```

- Each function takes **one argument**.
- Arguments must be provided in sequence.

Partial Application Example Partial application fixes some arguments but the resulting function can still take multiple arguments at once:

```
function volume(length, width, height) {
  return length * width * height;
}
function partialVolume(length) {
  return function (width, height) {
    return volume(length, width, height);
  };
}

const volumeLength2 = partialVolume(2);
console.log(volumeLength2(3, 4)); // 24
```

- Here, only `length` is fixed.
- The returned function takes the remaining arguments (`width`, `height`) together.

Key Differences Summarized

Feature	Currying	Partial Application
Function structure	Transforms to unary function chain	Fixes some args, fewer params
Arguments passed	One at a time, in sequence	Fixed args in any order, remaining args together
Flexibility	Less flexible, always unary	More flexible with argument positions
Usage	Great for composing small unary functions	Useful for creating specialized functions by pre-filling some arguments

11.2.1 Summary

- **Currying** converts a multi-argument function into a chain of unary functions.
- **Partial application** fixes some arguments of a function and returns a new function with fewer arguments.
- Both simplify function calls but serve different needs:
  - Use currying to enable elegant composition and unary function pipelines.
  - Use partial application to create specialized versions of functions quickly.

11.3 Building Curried Utilities and Function Templates

Currying becomes far more powerful when combined with reusable utilities and function templates. This section will guide you through creating a **generic curry helper** and building curried functions for common tasks, making your code modular, expressive, and easy to compose.

Writing a Generic `curry` Function

A generic `curry` function takes any multi-argument function and returns a curried version, allowing you to supply arguments one at a time:

```
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      // If enough arguments are provided, call the original function
      return fn(...args);
    } else {
      // Otherwise, return a function that takes the remaining arguments
      return function (...nextArgs) {
        return curried(...args, ...nextArgs);
      };
    }
```

```
  };
}

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

const curriedMultiply = curry(multiply);

console.log(curriedMultiply(2)(3)(4)); // 24
console.log(curriedMultiply(2, 3)(4)); // 24
console.log(curriedMultiply(2)(3, 4)); // 24
```

The `curry` utility lets you provide arguments one at a time or in groups, making it flexible and convenient.

Creating Curried Function Templates

Currying shines when building reusable templates for common operations:

Example 1: Curried Logger

```
const log = level => message => console.log(`[${level.toUpperCase()}] ${message}`);

const infoLog = log('info');
const errorLog = log('error');

infoLog('This is an informational message.');
errorLog('An error occurred!');
```

- Fixing the log level produces specialized loggers.
- Improves code reuse and readability.

Example 2: Curried Formatter

```
const format = prefix => suffix => str => `${prefix}${str}${suffix}`;

const emphasize = format('***')('***');
console.log(emphasize('Important'));  // ***Important***
```

- Currying breaks down string formatting into small reusable functions.

Example 3: Curried Math Operations

```
const add = a => b => a + b;
const subtract = a => b => b - a;  // Notice order reversed for demonstration

const addFive = add(5);
console.log(addFive(10)); // 15
```

Best Practices for Naming and Composition

- **Name curried functions clearly:** Use descriptive names like `add`, `multiply`, or `logWithLevel` to clarify their purpose.

- **Keep functions small and focused:** Each curried function should do one job, enhancing composability.
- **Use consistent parameter order:** Typically, place the most commonly fixed arguments first for ease of partial application.
- **Compose curried functions:** Combine small functions into pipelines for readable and declarative code.

Example of composing curried functions:

```
const trim = str => str.trim();
const toLower = str => str.toLowerCase();
const wrap = prefix => suffix => str => `${prefix}${str}${suffix}`;

const sanitizeAndWrap = str => wrap('<<')('>>')(toLower(trim(str)));

console.log(sanitizeAndWrap('  Hello World  ')); // <<hello world>>
```

11.3.1 Summary

- A **generic curry utility** transforms any function into a curried one, supporting flexible argument application.
- Curried **function templates** for logging, formatting, or math help create reusable, composable code blocks.
- Best practices include clear naming, focused functions, consistent argument ordering, and composing curried functions into pipelines.
- Mastering these patterns unlocks the full power of currying in your JavaScript functional programming toolkit.

Next, we will explore how to leverage these curried utilities in real-world functional programming scenarios.

Chapter 12.

Declarative Programming in Practice

1. Declarative vs Imperative Thinking
2. Rewriting Loops Declaratively
3. Practical Example: Declarative Data Filtering

12 Declarative Programming in Practice

12.1 Declarative vs Imperative Thinking

Programming styles generally fall into two broad categories: **imperative** and **declarative**. Understanding the difference between these styles is key to mastering functional programming in JavaScript.

Imperative Programming: How to Do It

Imperative programming is about **giving the computer explicit instructions**—step by step—on *how* to perform a task. You describe the control flow, manipulate variables, and update state as you go.

Example: Summing an array imperatively

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

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

console.log(sum); // 15
```

Here, we manually iterate through the array, update a running total, and control the loop explicitly.

Declarative Programming: What to Do

Declarative programming, in contrast, focuses on *what* you want to achieve, not *how* to do it. You express the logic at a higher level, often using expressions or functions that describe the desired outcome.

Example: Summing an array declaratively

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, n) => acc + n, 0);

console.log(sum); // 15
```

Using the `reduce()` method, you specify *what* you want (reduce the array to a single value by summing) without detailing the loop mechanics.

Another Side-by-Side Example: Filtering Even Numbers

Imperative style:

```
const numbers = [1, 2, 3, 4, 5];
const evens = [];
```

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

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

Declarative style:

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter(n => n % 2 === 0);

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

The declarative version clearly states the intent: filter the array for even numbers.

Why Prefer Declarative Code?

- **Readability:** Declarative code expresses the *intent* more clearly, making it easier to understand at a glance.
- **Maintainability:** With less boilerplate and explicit control flow, declarative code is easier to modify and extend.
- **Less Error-Prone:** Avoiding manual state mutation reduces bugs related to variables and side effects.
- **Composability:** Declarative functions compose naturally, enabling reusable and modular code.

12.1.1 Summary

- **Imperative programming** details *how* to do things via explicit control flow and state changes.
- **Declarative programming** focuses on *what* to do by describing outcomes through expressions or function calls.
- Functional programming in JavaScript encourages declarative styles using methods like `map()`, `filter()`, and `reduce()`.
- Writing declarative code improves readability, reduces errors, and helps build maintainable, composable programs.

Next, we'll explore how to rewrite common loops declaratively, moving from verbose control structures to expressive, functional code.

12.2 Rewriting Loops Declaratively

Traditional loops like `for` and `while` are the backbone of imperative programming, where you explicitly tell the program how to iterate and manipulate data. In functional programming, however, we replace these loops with **declarative array methods** like `map()`, `filter()`, and `reduce()`. These methods express *what* you want to do with the data instead of *how* to do it.

From `for` Loop to `map()`

Imperative example: Transforming an array by doubling each number

```
const numbers = [1, 2, 3, 4];
const doubled = [];

for (let i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2);
}

console.log(doubled); // [2, 4, 6, 8]
```

Declarative equivalent using `map()`:

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

console.log(doubled); // [2, 4, 6, 8]
```

With `map()`, you describe the transformation of each element without worrying about loop counters or pushing to an array.

From `for` Loop to `filter()`

Imperative example: Filtering even numbers from an array

```
const numbers = [1, 2, 3, 4, 5];
const evens = [];

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

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

Declarative equivalent using `filter()`:

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter(n => n % 2 === 0);

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

The `filter()` method clearly states the intent: keep only the elements matching a condition.

From Loop to `reduce()`

Imperative example: Summing numbers in an array

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

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

console.log(sum); // 15
```

Declarative equivalent using `reduce()`:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, n) => acc + n, 0);

console.log(sum); // 15
```

`reduce()` aggregates values by applying a function cumulatively, abstracting away the loop control.

Benefits of This Shift

- **Immutability:** Declarative methods return new arrays or values instead of modifying existing ones, preventing unintended side effects.
- **Clearer intent:** Methods like `map` and `filter` communicate your goals directly.
- **Less boilerplate:** You write less code, reducing room for errors related to indexing and manual state management.
- **Chainability:** These methods can be chained to build expressive data transformation pipelines.

12.2.1 Summary

Rewriting traditional loops using declarative array methods improves your code by:

- Focusing on *what* transformation you want rather than *how* to do it.
- Encouraging immutability by avoiding mutation of original data.
- Producing more readable, maintainable, and composable code.

In the next section, we'll see a practical example of declarative data filtering that combines these techniques to solve a real-world problem.

12.3 Practical Example: Declarative Data Filtering

To see declarative programming in action, let's work with a more complex dataset—an array of user objects—and filter it based on multiple conditions using reusable, composable predicates and the `filter()` method.

The Dataset

Imagine you have an array of user objects like this:

```
const users = [
  { id: 1, name: 'Alice', age: 25, active: true, role: 'admin' },
  { id: 2, name: 'Bob', age: 30, active: false, role: 'user' },
  { id: 3, name: 'Charlie', age: 35, active: true, role: 'user' },
  { id: 4, name: 'Dana', age: 28, active: true, role: 'moderator' },
  { id: 5, name: 'Eli', age: 40, active: false, role: 'admin' },
];
```

Step 1: Define Reusable Predicate Functions

Instead of writing complex inline conditions, define small, reusable predicate functions that return `true` or `false` based on the user properties:

```
const isActive = user => user.active === true;
const isAdmin = user => user.role === 'admin';
const isUser = user => user.role === 'user';
const isOlderThan = ageLimit => user => user.age > ageLimit;
```

These predicates are **pure functions** that clearly express one filtering criterion each.

Step 2: Compose Multiple Conditions with `filter()`

Now, suppose you want to find **all active users who are older than 27**. You can chain predicates inside a single `filter()` call:

```
const filteredUsers = users.filter(user => isActive(user) && isOlderThan(27)(user));

console.log(filteredUsers);
```

Output:

```
[
  { id: 3, name: 'Charlie', age: 35, active: true, role: 'user' },
  { id: 4, name: 'Dana', age: 28, active: true, role: 'moderator' }
]
```

Step 3: Build a Helper to Combine Predicates (Optional)

For cleaner and more reusable filtering logic, you can create a small utility to combine predicates:

```
const allPass = predicates => value => predicates.every(predicate => predicate(value));

const isActiveAndOlderThan27 = allPass([isActive, isOlderThan(27)]);
const activeAdmins = allPass([isActive, isAdmin]);

const filteredUsers1 = users.filter(isActiveAndOlderThan27);
const filteredUsers2 = users.filter(activeAdmins);

console.log(filteredUsers1);
console.log(filteredUsers2);
```

This helper accepts an array of predicate functions and returns a function that returns `true` only if **all** predicates pass.

Full runnable code:

```
// Dataset
const users = [
  { id: 1, name: 'Alice', age: 25, active: true, role: 'admin' },
  { id: 2, name: 'Bob', age: 30, active: false, role: 'user' },
  { id: 3, name: 'Charlie', age: 35, active: true, role: 'user' },
  { id: 4, name: 'Dana', age: 28, active: true, role: 'moderator' },
  { id: 5, name: 'Eli', age: 40, active: false, role: 'admin' },
];

// Step 1: Reusable predicates
const isActive = user => user.active === true;
const isAdmin = user => user.role === 'admin';
const isUser = user => user.role === 'user';
const isOlderThan = ageLimit => user => user.age > ageLimit;

// Step 2: Filter with inline composed conditions
const filteredUsers = users.filter(user => isActive(user) && isOlderThan(27)(user));

console.log('Active users older than 27:');
console.log(filteredUsers);

// Step 3: Optional helper to combine predicates
const allPass = predicates => value =>
  predicates.every(predicate => predicate(value));

const isActiveAndOlderThan27 = allPass([isActive, isOlderThan(27)]);
const activeAdmins = allPass([isActive, isAdmin]);

const filteredUsers1 = users.filter(isActiveAndOlderThan27);
const filteredUsers2 = users.filter(activeAdmins);

console.log('\nUsing allPass:');
console.log('Active & older than 27:');
console.log(filteredUsers1);

console.log('\nActive Admins:');
console.log(filteredUsers2);
```

Why Declarative Filtering Shines

- **Clarity:** Each predicate has a clear, single responsibility.
- **Reusability:** You can reuse and combine predicates in different contexts without rewriting logic.
- **Testability:** Pure predicate functions are easy to test independently.
- **No Side Effects:** Original data is never mutated; filtering returns new arrays.
- **Expressiveness:** The intent behind each filter is explicit and easy to understand.

12.3.1 Summary

In this example, we used declarative array methods and pure, composable predicate functions to filter complex data clearly and flexibly. This approach reduces boilerplate and hidden state management, making your code easier to maintain and reason about.

By embracing declarative programming, your JavaScript becomes more expressive, modular, and aligned with functional programming best practices.

Chapter 13.

Point-Free Style

1. What Is Point-Free Programming?
2. When and How to Use It
3. Refactoring for Simplicity and Reusability

13 Point-Free Style

13.1 What Is Point-Free Programming?

Point-free programming, also known as **tacit programming**, is a style of writing functions **without explicitly mentioning the arguments** on which they operate. Instead of focusing on the data (the "points"), you define functions by **composing other functions**.

This style has its roots in mathematical function composition and category theory, where the emphasis is on combining functions to build more complex behavior without naming intermediate variables.

Why Point-Free?

The key idea is to shift focus away from *how* data flows through functions and instead focus on *what* transformations are composed. This promotes code that is more declarative, modular, and often easier to reason about.

Example: Normal vs Point-Free Style

Consider a simple function that trims whitespace and converts a string to lowercase:

Normal style (with explicit arguments):

```
const normalize = str => str.trim().toLowerCase();

console.log(normalize("  Hello World!  ")); // "hello world!"
```

Point-free style (composing functions):

```
const trim = str => str.trim();
const toLowerCase = str => str.toLowerCase();

const compose = (f, g) => x => f(g(x));
const normalize = compose(toLowerCase, trim);

console.log(normalize("  Hello World!  ")); // "hello world!"
```

In the point-free version, `normalize` is defined as the composition of `toLowerCase` after `trim`, without explicitly mentioning the argument `str`.

Another Example: Adding 1 and Doubling

Normal style:

```
const addOneThenDouble = x => (x + 1) * 2;

console.log(addOneThenDouble(4)); // 10
```

Point-free style:

```
const addOne = x => x + 1;
const double = x => x * 2;

const compose = (f, g) => x => f(g(x));
const addOneThenDouble = compose(double, addOne);

console.log(addOneThenDouble(4)); // 10
```

Again, the argument `x` is not explicitly mentioned in the final function.

When Does Point-Free Style Help?

- **Improves abstraction:** By focusing on composition, it highlights *what* is done rather than *how*.
- **Reduces boilerplate:** No need to mention or manage arguments explicitly.
- **Promotes reusability:** Encourages breaking code into small, composable functions.

When to Avoid It?

- **Complexity:** Overuse of point-free style can make code hard to read, especially for beginners or when functions become too nested.
- **Debugging difficulty:** Without named arguments, it might be less clear where errors or unexpected values originate.
- **Loss of clarity:** Sometimes explicitly naming arguments makes the intent more obvious.

13.1.1 Summary

Point-free programming is a powerful functional style that defines functions by composing other functions without naming their inputs explicitly. It aligns well with functional programming principles of composition and modularity, offering clean, concise code when used judiciously. However, balancing clarity and abstraction is essential to keep your code readable and maintainable.

13.2 When and How to Use It

Point-free programming is a powerful technique that shines in certain practical scenarios, especially when building **function pipelines** or composing small, reusable utilities. However, like any tool, it's important to know **when to use it—and when to avoid it**—to maintain code clarity and readability.

When to Use Point-Free Style

- **Building Pipelines and Compositions:** When you want to express a sequence of transformations clearly, point-free style lets you focus on *what* functions are composed rather than *how* data flows between them. For example, chaining data transformations with `compose()` or `pipe()` is naturally point-free.
- **Composing Small Utilities:** If you have simple, single-argument functions, composing them without explicitly handling arguments leads to concise, reusable code.
- **Avoiding Boilerplate:** For straightforward operations, point-free style reduces the clutter of parameter names, making your intent clearer.

When to Avoid Point-Free Style

- **Complex Logic or Multiple Arguments:** When functions take multiple parameters or involve conditional logic, point-free style can become obscure and harder to understand.
- **Code Readability Concerns:** If the resulting function composition is deeply nested or unintuitive, explicitly naming parameters may improve clarity.
- **Debugging Difficulty:** Without explicit parameters, tracing values during debugging can be trickier.

How to Transform Regular Functions into Point-Free Style

Let's look at a step-by-step example.

Step 1: Regular function with explicit arguments

```
const greet = name => `Hello, ${name.trim()}!`;
```

Step 2: Extract smaller functions

```
const trim = str => str.trim();
const greet = name => `Hello, ${name}!`;
```

Step 3: Compose functions (still mentioning arguments)

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

Step 4: Refactor to point-free style by composing functions

```
const trim = str => str.trim();
const greetMessage = name => `Hello, ${name}!`;

const compose = (f, g) => x => f(g(x));

const greet = compose(greetMessage, trim);

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

Common Pitfalls to Avoid

- **Over-Composition:** Chaining too many composed functions can reduce readability. Break long compositions into named intermediate steps.
- **Ignoring Multiple Arguments:** Point-free style works best with unary (single-argument) functions. For functions with multiple parameters, consider partial application or other patterns.
- **Losing Context:** Over-abstracting with point-free style can make it hard to follow data flow, especially for newcomers.

13.2.1 Summary

Use point-free style to write clean, reusable, and expressive code—especially for pipelines and composing small utilities. But always weigh abstraction against readability. When code becomes too terse or complex, adding explicit arguments back in can improve understanding and maintainability.

Balance is key: write point-free where it makes your code clearer, but don't hesitate to be explicit when clarity calls for it.

13.3 Refactoring for Simplicity and Reusability

Refactoring your code into **point-free style** can greatly enhance its simplicity, modularity, and reusability. By focusing on composing functions rather than explicitly handling data or parameters, you encourage thinking about your code as a series of **transformations** rather than step-by-step manipulations.

Example 1: Verbose Function Refactored into Point-Free Style

Imagine a function that formats a user's full name by trimming whitespace, converting to lowercase, and then capitalizing the first letter of each word:

```
function formatName(name) {
  const trimmed = name.trim();
  const lowercased = trimmed.toLowerCase();
  const capitalized = lowercased
    .split(' ')
    .map(word => word.charAt(0).toUpperCase() + word.slice(1))
    .join(' ');
  return capitalized;
}

console.log(formatName('  aLiCe joHNsON  ')); // "Alice Johnson"
```

Though straightforward, this function has explicit intermediate variables and manipulates data step-by-step.

Refactor with Small, Reusable Functions

Break down each step into smaller reusable functions:

```
const trim = str => str.trim();
const toLowerCase = str => str.toLowerCase();
const capitalize = str => str.charAt(0).toUpperCase() + str.slice(1);
const capitalizeWords = str =>
  str.split(' ').map(capitalize).join(' ');
```

Use `compose()` for Point-Free Style

Now, compose these functions into a clean, point-free pipeline:

```
const compose = (f, g) => x => f(g(x));
const trim = str => str.trim();
const toLowerCase = str => str.toLowerCase();
const capitalize = str => str.charAt(0).toUpperCase() + str.slice(1);
const capitalizeWords = str =>
  str.split(' ').map(capitalize).join(' ');

const formatName = compose(capitalizeWords, compose(toLowerCase, trim));

console.log(formatName('  aLiCe joHNsON  ')); // "Alice Johnson"
```

Here, `formatName` no longer explicitly mentions its argument. It simply **composes** three transformations.

Example 2: Reusable Logging Wrapper

You can also refactor logging or other cross-cutting concerns as reusable wrappers:

```
const log = fn => (...args) => {
  console.log('Input:', args);
  const result = fn(...args);
  console.log('Output:', result);
  return result;
};

// Original function
const square = x => x * x;

// Wrap with logging in point-free style
const loggedSquare = log(square);

loggedSquare(5);
// Input: [5]
// Output: 25
```

This higher-order function is reusable and keeps the core logic separate from side effects.

Why This Matters

- **Simplicity:** Functions focus on a single transformation, avoiding cluttered variable assignments.
- **Reusability:** Small functions like `trim` or `capitalize` can be reused elsewhere.
- **Modularity:** Composed functions can be combined and rearranged easily.
- **Declarative Thinking:** You define *what* transformations happen, not *how* data is passed around.

13.3.1 Summary

Refactoring into point-free style encourages you to build programs from **small, composable building blocks**. Using helpers like `compose` or `pipe`, your code becomes clearer and easier to maintain, highlighting data transformations instead of data handling. This shift is at the heart of functional programming's power and elegance.

Chapter 14.

Function Utilities and Combinators

1. Identity, Constant, Flip, and Tap
2. Useful Functional Combinators (e.g. `once`, `memoize`, `throttle`)
3. Building Your Own Utility Functions

14 Function Utilities and Combinators

14.1 Identity, Constant, Flip, and Tap

In functional programming, **combinators** are small utility functions that help you build more complex behavior by manipulating or controlling functions themselves. Four foundational combinators—**identity**, **constant**, **flip**, and **tap**—are simple yet incredibly useful in crafting clear, reusable functional code.

Identity

The **identity** function simply returns whatever input it receives, unchanged.

```
const identity = x => x;

// Usage
console.log(identity(42)); // 42
console.log(identity("hello")); // "hello"
```

Purpose & Use Cases:

- Acts as a default or fallback function.
- Useful in function compositions where a no-op function is needed.
- Helps in debugging by passing data through unchanged.

Constant

The **constant** function takes a value and returns a new function that always returns that same value, regardless of its input.

```
const constant = value => () => value;

// Usage
const alwaysTrue = constant(true);
console.log(alwaysTrue());      // true
console.log(alwaysTrue(123));   // true, ignores input
```

Purpose & Use Cases:

- Useful when you want a function that ignores its arguments but returns a fixed value.
- Can simplify defaults or placeholders in higher-order functions.

Flip

The **flip** function takes a binary function and returns a new function with the order of its first two arguments reversed.

```
const flip = fn => (a, b, ...rest) => fn(b, a, ...rest);

// Usage
const subtract = (a, b) => a - b;
```

```
const flippedSubtract = flip(subtract);

console.log(subtract(10, 5));        // 5
console.log(flippedSubtract(10, 5)); // -5
```

Purpose & Use Cases:

- Useful when you want to change the argument order to fit a particular interface or composition.
- Helps make functions more composable by aligning their argument orders.

Tap

The **tap** function allows you to perform a side effect (like logging) without modifying the value passing through a function chain.

```
const tap = fn => x => {
  fn(x);
  return x;
};

// Usage
const double = x => x * 2;

const process = x =>
  tap(console.log)(double(x));

console.log(process(5)); // Logs: 10, then outputs: 10
```

Purpose & Use Cases:

- Great for debugging or logging inside a chain of function calls without disrupting the data flow.
- Helps keep pure functions pure by isolating side effects.

14.1.1 Summary

These four foundational combinators are building blocks for more expressive functional programming:

Function	Purpose	Typical Use Case
`identity`	Returns input unchanged	Default functions, debugging
`constant`	Returns a fixed value	Defaults, ignoring arguments
`flip`	Reverses first two arguments	Argument order adjustment
`tap`	Runs side effect, returns input	Debugging, logging in pipelines

Mastering these simple utilities will help you write cleaner, more modular, and more reusable

JavaScript functional code.

14.2 Useful Functional Combinators (e.g. `once`, `memoize`, `throttle`)

Beyond basic combinators, functional programming provides **utility combinators** that modify how functions behave, helping manage performance, side effects, and control flow. Three common and powerful combinators are **`once`**, **`memoize`**, and **`throttle`**.

`once`

The `once` combinator ensures that a given function runs **only once**. Subsequent calls return the result of the first call, preventing repeated execution.

```
const once = fn => {
  let called = false;
  let result;
  return (...args) => {
    if (!called) {
      result = fn(...args);
      called = true;
    }
    return result;
  };
};

// Usage
const initialize = once(() => {
  console.log('Initialization done');
  return 42;
});

console.log(initialize()); // Logs "Initialization done", returns 42
console.log(initialize()); // Returns 42, does not log again
```

Use Cases:

- Initialize resources only once (e.g., setting up event listeners).
- Avoid repeated costly or side-effectful operations.

`memoize`

`memoize` caches the results of function calls based on their arguments, so repeated calls with the same inputs return the cached result instead of recomputing. This improves performance for expensive or recursive functions.

```
const memoize = fn => {
  const cache = new Map();
  return (...args) => {
```

```
    const key = JSON.stringify(args);
    if (cache.has(key)) {
      return cache.get(key);
    }
    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
};

// Usage: Memoized Fibonacci
const fib = memoize(n => {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
});

console.log(fib(40)); // Efficiently computes Fibonacci
```

Use Cases:

- Optimize recursive or CPU-intensive calculations.
- Reduce duplicate work in pure functions.

`throttle`

The `throttle` combinator limits how often a function can be called over time. It ensures the function runs at most once per specified delay interval, useful for performance optimization in events like scrolling or resizing.

```
const throttle = (fn, delay) => {
  let lastCall = 0;
  return (...args) => {
    const now = Date.now();
    if (now - lastCall >= delay) {
      lastCall = now;
      return fn(...args);
    }
  };
};

// Usage: Throttle a window resize handler
const onResize = () => console.log('Resized!');
window.addEventListener('resize', throttle(onResize, 1000));
```

Use Cases:

- Limit high-frequency event handler calls (scroll, resize, keypress).
- Prevent overwhelming APIs or UI updates.

14.2.1 Why Use These Combinators?

These utilities enhance your code by:

- **Performance Optimization:** `memoize` avoids unnecessary recomputation, and `throttle` reduces excessive function calls.
- **Control Flow Management:** `once` ensures side-effectful initialization happens a single time.
- **Code Clarity and Reusability:** Wrapping logic in composable combinators separates concerns and makes your functions easier to maintain and test.

14.2.2 Summary Table

Combinator	Purpose	Example Use Case
`once`	Run a function only once	Initialization
`memoize`	Cache results of expensive calls	Recursive Fibonacci or API caching
`throttle`	Limit function call rate	Scroll or resize event handling

Mastering these combinators equips you with powerful tools to write **efficient, controlled, and modular** functional JavaScript code.

14.3 Building Your Own Utility Functions

Creating your own **functional utilities** and combinators is a powerful way to tailor reusable, composable building blocks that fit your projects' needs. This section guides you through designing utilities that follow **functional programming best practices**: immutability, purity, and predictable interfaces.

Best Practices for Utility Functions

Before diving into code, keep these principles in mind:

- **Pure Functions:** Your utilities should avoid side effects, relying only on their inputs to produce outputs.
- **Immutability:** Don't mutate arguments or external state; always return new values when necessary.
- **Predictability:** Functions should have clear, consistent behavior with well-defined inputs and outputs.
- **Composability:** Design utilities so they can easily combine with others (e.g., by accepting and returning functions).

- **Documentation & Testing:** Write clear comments and unit tests to ensure your utilities are maintainable and bug-free.

Building a Simple `debounce` Utility

A **debounce** function delays invoking a target function until after a specified wait time has elapsed since the last time it was called. This is useful for rate-limiting frequent events (like typing or window resizing).

Step 1: Define the function signature Our `debounce` will:

- Accept a function `fn` to debounce
- Accept a delay time `wait` in milliseconds
- Return a new debounced function

```
function debounce(fn, wait) {
  let timeoutId;

  // Return a debounced version of fn
  return function debounced(...args) {
    // Clear the previous timer if function called again quickly
    clearTimeout(timeoutId);

    // Schedule fn to run after `wait` milliseconds
    timeoutId = setTimeout(() => {
      fn(...args);
    }, wait);
  };
}
```

Step 2: Implement the debounce logic

```
const logResize = () => console.log('Window resized!');
const debouncedResize = debounce(logResize, 300);

window.addEventListener('resize', debouncedResize);
```

Step 3: Usage example In this example, `logResize` is only called **300 milliseconds** after the user stops resizing the window, preventing excessive function calls.

Why This Is Functional

- The `debounce` function **does not mutate** any external state outside its closure (except `timeoutId`, which is internal and encapsulated).
- It **returns a new function** without side effects on its own.
- It provides a **predictable interface**: calling the debounced function schedules the original function's execution.

Extending Further: Retry Utility Example

Another useful combinator is a **retry** function that attempts to call an asynchronous function multiple times before failing.

```
function retry(fn, retries = 3, delay = 1000) {
  return async function retried(...args) {
    for (let attempt = 1; attempt <= retries; attempt++) {
      try {
        return await fn(...args);
      } catch (err) {
        if (attempt === retries) throw err;
        await new Promise(resolve => setTimeout(resolve, delay));
      }
    }
  };
}
```

Testing and Documentation

- Write unit tests covering edge cases, such as immediate repeated calls for `debounce` or error handling for `retry`.
- Document parameters, return values, and side effects clearly.
- Keep your utilities **small and focused** for easier maintenance.

14.3.1 Summary

Building your own functional utilities helps you:

- Customize behavior tailored to your application
- Reuse code with confidence due to purity and immutability
- Create predictable, testable building blocks for complex function compositions

With practice, you'll find yourself creating elegant combinators that simplify and strengthen your JavaScript functional programming toolkit.

Chapter 15.

Functional Programming with Lodash/fp and Ramda

1. Overview of Lodash/fp and Ramda
2. Practical Differences from Vanilla JS
3. Real Examples: Currying, Deep Transformations, and Pipelining

15 Functional Programming with Lodash/fp and Ramda

15.1 Overview of Lodash/fp and Ramda

In the JavaScript functional programming ecosystem, **Lodash/fp** and **Ramda** stand out as two of the most popular libraries designed to make functional patterns easier and more consistent.

What Are Lodash/fp and Ramda?

Both libraries provide **functional utilities** that extend JavaScript's built-in capabilities, with a strong focus on:

- **Immutability:** Functions do not mutate inputs but return new values, helping prevent side effects.
- **Currying by Default:** Functions are automatically curried, meaning you can partially apply arguments and build pipelines easily.
- **Functional Style:** They emphasize declarative, composable utilities that simplify common tasks like data transformation, function composition, and iteration.

Design Goals and Philosophy

- **Lodash/fp** is a functional programming variant of the widely used Lodash library. It transforms Lodash's methods into **auto-curried**, **immutable**, and **data-last** forms, enabling better function composition and point-free style.
- **Ramda** was built from the ground up for functional programming. It uses **currying by default**, **immutable data handling**, and encourages a **point-free style**, often making function composition more straightforward and expressive.

Key Differences

Feature	Lodash/fp	Ramda
Origin	FP variant of Lodash utility library	Designed explicitly for FP from start
Currying	Auto-curried	Auto-curried
Argument order	Data-last, facilitating composition	Data-last, facilitating composition
Ecosystem and Usage	Popular in projects already using Lodash	Popular among pure FP advocates
Learning curve	Easier for Lodash users	Slightly steeper, more FP-centric

Installation and Basic Usage

To add these libraries to your project, use npm or yarn:

```
npm install lodash lodash/fp
npm install ramda
```

In your JavaScript code:

```
// Lodash/fp example
import { map, filter } from 'lodash/fp';

// Ramda example
import * as R from 'ramda';
```

Both allow building expressive, reusable functions with ease, for example:

```
// Lodash/fp: filter and map
const result = map(x => x * 2, filter(x => x > 10, [5, 15, 20]));

// Ramda: filter and map
const result = R.map(x => x * 2, R.filter(x => x > 10, [5, 15, 20]));
```

Summary

Lodash/fp and **Ramda** provide robust, functional-first utilities that encourage immutable, curried, and composable code in JavaScript. Choosing between them depends on your project's background and preferences, but both significantly ease working in a functional style beyond vanilla JavaScript.

15.2 Practical Differences from Vanilla JS

When working with functional programming in JavaScript, you can certainly rely on **vanilla JS** features such as arrow functions, array methods (`map`, `filter`, `reduce`), and closures. However, libraries like **Lodash/fp** and **Ramda** offer enhanced utilities and patterns that make functional programming more powerful, concise, and consistent. Let's explore the key practical differences.

Automatic Currying

- **Vanilla JS:** Functions do not support currying by default. You must manually write functions to return other functions or use helper utilities.

```
// Vanilla JS function: no automatic currying
function add(a, b) {
  return a + b;
}

// Manual currying
const addCurried = a => b => a + b;

addCurried(2)(3); // 5
```

- **Lodash/fp and Ramda:** Functions are **auto-curried**, meaning you can call them with fewer arguments than expected, and they return a new function waiting for the rest.

```
import { add } from 'lodash/fp';
import * as R from 'ramda';

const addFive = add(5);   // Partially applied function
console.log(addFive(10)); // 15

const addFiveR = R.add(5);
console.log(addFiveR(10)); // 15
```

Argument Order (Data-Last)

- **Vanilla JS:** The data argument usually comes first, and callback or transform functions come last (e.g., `array.map(callback)`).
- **Lodash/fp and Ramda:** Functions use **data-last** argument order to facilitate easier composition and partial application.

```
// Vanilla JS
const doubled = [1, 2, 3].map(x => x * 2);

// Lodash/fp and Ramda
import { map } from 'lodash/fp';
import * as R from 'ramda';

const double = x => x * 2;

// Data-last, so data is last argument
const doubledLodash = map(double)([1, 2, 3]);
const doubledRamda = R.map(double)([1, 2, 3]);
```

This ordering allows easy creation of reusable functions and pipelines without needing to specify data upfront.

Immutability Guarantees

- **Vanilla JS:** Immutability depends on how you write your code; native methods like `map` and `filter` return new arrays, but others like `push` mutate arrays.
- **Lodash/fp and Ramda:** All utility functions are designed to be **immutable**, never mutating the inputs, which makes data handling safer and side-effect free.

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

// Vanilla JS map is immutable
const newArr = arr.map(x => x + 1);
console.log(arr);    // [1, 2, 3]
console.log(newArr); // [2, 3, 4]

// Lodash/fp and Ramda follow immutability consistently even in complex utilities
```

Richer and More Consistent Utility Sets

- **Vanilla JS:** The built-in methods focus mainly on arrays and objects, lacking some advanced functional utilities like `compose`, `pipe`, deep cloning, or currying helpers.
- **Lodash/fp and Ramda:** Provide a **comprehensive set** of functional utilities like:
  - Function composition (`compose`, `pipe`)
  - Deep data transformations
  - Currying helpers
  - Object utilities with functional flavor (immutable setters/getters)

```
// Compose functions in Ramda
const trim = str => str.trim();
const toLower = str => str.toLowerCase();

const normalize = R.compose(toLower, trim);
console.log(normalize("  Hello World  ")); // "hello world"
```

Vanilla JS requires manual chaining or nested calls, which can be less readable.

Summary Comparison

Feature	Vanilla JS	Lodash/fp & Ramda
Currying	Manual, requires custom code	Auto-curried by default
Argument order	Data-first (e.g., array methods)	Data-last for better composition
Immutability	Depends on method and usage	Guaranteed immutable operations
Utility richness	Limited to core JS methods	Extensive, consistent functional utilities
Composition	Nested calls or chaining	`compose()`, `pipe()` utilities

By adopting **Lodash/fp** or **Ramda**, you gain expressive, composable, and predictable functional tools that extend vanilla JavaScript's functional capabilities — enabling cleaner, more maintainable codebases.

15.3 Real Examples: Currying, Deep Transformations, and Pipelining

Lodash/fp and Ramda shine in real-world functional programming tasks by simplifying complex operations like currying, deep data transformations, and composing pipelines. Let's explore these with concrete, runnable examples.

15.3.1 Creating Curried Functions

Both Lodash/fp and Ramda automatically curry functions, allowing you to partially apply arguments for flexible reuse.

```
import { multiply } from 'lodash/fp';
import * as R from 'ramda';

// Lodash/fp curried multiply
const multiplyBy2 = multiply(2);
console.log(multiplyBy2(5)); // 10

// Ramda curried multiply
const multiplyBy3 = R.multiply(3);
console.log(multiplyBy3(5)); // 15
```

With currying, you can create specialized functions easily without repeating arguments, enabling composition and cleaner code.

15.3.2 Deep Transformations on Nested Objects or Arrays

Vanilla JavaScript requires verbose, imperative code to update deeply nested data immutably. Ramda and Lodash/fp provide utilities like `assocPath`, `update`, and `set` that handle this declaratively.

Example: Updating a deeply nested property

```
import * as R from 'ramda';

const user = {
  name: 'Alice',
  address: {
    city: 'Seattle',
    zip: 98101,
  },
};

// Update city immutably using Ramda's assocPath
const updatedUser = R.assocPath(['address', 'city'], 'Portland', user);

console.log(updatedUser.address.city); // "Portland"
console.log(user.address.city);        // "Seattle" (original unchanged)
```

Similarly, Lodash/fp offers `set` and `update`:

```
import { set } from 'lodash/fp';

const updatedUser2 = set(['address', 'city'], 'Portland', user);

console.log(updatedUser2.address.city); // "Portland"
```

These utilities help manage nested structures without mutation, making state updates safer and clearer.

15.3.3 Building Data Pipelines with `compose()` or `pipe()`

Pipelining lets you combine multiple small, reusable functions into clear data transformation flows.

Using Ramda's `pipe()` (left-to-right):

```
import * as R from 'ramda';

const trim = str => str.trim();
const toLower = str => str.toLowerCase();
const removeExclamation = str => str.replace(/!/g, '');

const normalize = R.pipe(trim, toLower, removeExclamation);

console.log(normalize('  Hello WORLD!!!  ')); // "hello world"
```

Using Lodash/fp's `flow()` (similar to `pipe()`):

```
import { flow, trim, toLower, replace } from 'lodash/fp';

const normalize2 = flow(
  trim,
  toLower,
  str => replace(/!/g, '', str)
);

console.log(normalize2('  Hello WORLD!!!  ')); // "hello world"
```

You avoid deeply nested function calls (`f(g(h(x)))`) and make the data transformations declarative and easy to read.

15.3.4 Benefits:

- **Expressiveness:** Compose complex operations succinctly.
- **Reusability:** Small functions easily reused and combined.
- **Immutability:** Built-in guarantees prevent unintended mutations.
- **Readability:** Pipelines and currying create clear intent and flow.

15.3.5 Drawbacks:

- **Learning Curve:** Some syntax and concepts (currying, pipelines) can be unfamiliar.
- **Debugging:** Tracing through composed pipelines may be tricky without good tooling.
- **Bundle Size:** Adding Lodash/fp or Ramda increases dependency size (mitigated by tree-shaking).

15.3.6 Summary

By leveraging Lodash/fp and Ramda, you can write clean, maintainable functional JavaScript that excels at handling:

- **Curried functions** for flexible argument handling,
- **Deep, immutable transformations** on complex nested data, and
- **Readable pipelines** that describe *what* happens to data step-by-step.

These libraries help you unlock the full power of functional programming beyond vanilla JavaScript capabilities.

Chapter 16.

Monads, Functors, and Maybes

1. Introduction to Functors, Monads, and `Maybe` Patterns
2. Safe Composition with `Maybe` and `Either`
3. Practical Example: Avoiding Null Errors Functionally

16 Monads, Functors, and Maybes

16.1 Introduction to Functors, Monads, and `Maybe` Patterns

Functional programming introduces powerful abstractions that help manage complexity and side effects in a clean, predictable way. Three key concepts you'll encounter are **functors**, **monads**, and the **`Maybe` pattern**.

What Is a Functor?

A **functor** is a type (or container) that holds a value and provides a way to apply a function to that value *without* changing the structure of the container.

In JavaScript terms, think of an array as a functor: you can use `.map()` to apply a function to each element, and the array structure remains intact.

Functor law: If you map the identity function over a functor, you get the same functor back.

```
// Array as a functor
const numbers = [1, 2, 3];

// Map doubles each element but keeps the array structure
const doubled = numbers.map(x => x * 2);
console.log(doubled); // [2, 4, 6]
```

Diagram:

```
Functor Container
+-------------+
| value(s)    |
+-------------+
      |
      v  map(f)
+-------------+
| f(value(s)) |
+-------------+
```

What Is a Monad?

A **monad** builds on functors by providing a way to chain operations that return wrapped values (functors), handling the "context" automatically. This allows you to sequence computations that might have side effects, errors, or asynchronous behavior, all while keeping your code clean and declarative.

In JavaScript, promises are monads: they let you chain asynchronous computations without deeply nested callbacks.

Monads must implement two main operations:

- **`of` (or `return`)**: wraps a value in the monad context
- **`chain` (or `flatMap`)**: applies a function that returns a monad, flattening the result

Example (simplified):

```
// Simplified Promise-like monad chaining
Promise.resolve(2)
  .then(x => Promise.resolve(x * 3)) // returns a new Promise (monad)
  .then(console.log); // 6
```

The `Maybe` Monad: Handling Optional Values Safely

`Maybe` (also called `Option`) is a special kind of monad that represents a value that might be there (`Just` or `Some`) or might be absent (`Nothing` or `None`), eliminating null checks and exceptions.

Instead of manually checking for null or undefined, `Maybe` wraps the value and provides safe methods to work with it.

Simple implementation:

```
class Maybe {
  constructor(value) {
    this.value = value;
  }

  static of(value) {
    return new Maybe(value);
  }

  isNothing() {
    return this.value === null || this.value === undefined;
  }

  map(fn) {
    if (this.isNothing()) {
      return Maybe.of(null);
    }
    return Maybe.of(fn(this.value));
  }

  // ... additional methods like chain/flatMap could be added
}

const result = Maybe.of('Functional Programming')
  .map(str => str.toUpperCase())
  .map(str => str.split(' '));

console.log(JSON.stringify(result,null,2)); // Maybe { value: [ 'FUNCTIONAL', 'PROGRAMMING' ] }
```

If the initial value were `null`, all `map` calls would safely return a `Maybe` containing `null` without throwing errors.

16.1.1 Why These Concepts Matter

- **Functors** let you apply transformations cleanly inside containers (arrays, promises, Maybes).
- **Monads** let you chain computations while managing context like failure, state, or async.
- `Maybe` helps avoid null-related bugs by representing optional data explicitly.

16.1.2 Summary Diagram

```
Value ----> Functor (supports map)
              |
              v
            Monad (supports chain)
              |
              v
         Maybe (monad for optional values)
```

16.1.3 Recap with JavaScript Developers in Mind

Concept	What it is	JS Example	Purpose
Functor	Container that can be mapped over	Array `.map()`	Apply functions while preserving structure
Monad	Functor + chaining (`chain/flatMap`)	Promise `.then()`	Compose sequences of context-aware computations
Maybe	Monad for optional values	Custom `Maybe` class	Avoid null errors with safe chaining

Understanding functors, monads, and `Maybe` prepares you to write more robust, declarative, and error-resistant JavaScript functional code.

16.2 Safe Composition with `Maybe` and `Either`

In functional programming, **safe composition** means combining multiple operations that might fail or produce no result — without throwing exceptions or crashing your program.

Two powerful monads that help achieve this in JavaScript are **`Maybe`** and **`Either`**.

Why Use `Maybe` and `Either`?

- **`Maybe`** represents optional values — either a value **exists** (`Just` or `Some`) or it **does not** (`Nothing` or `None`). It helps avoid null/undefined errors by safely encapsulating absence.
- **`Either`** represents a computation that can **succeed** or **fail**, carrying either a success value (`Right`) or an error/info value (`Left`). This lets you handle failure cases explicitly with additional context.

Both monads enable *safe chaining* of operations that might fail, without exceptions disrupting the flow.

Difference Between `Maybe` and `Either`

Aspect	Maybe	Either
Purpose	Handle optional/missing values	Handle success/failure with error info
States	`Just(value)` or `Nothing`	`Right(value)` or `Left(error)`
Error Info	No error details	Can carry detailed error or failure
Use Case Example	Optional user input or data	API response success/failure handling

Safe Chaining with `Maybe`

Imagine reading a deeply nested user property that may or may not exist, which often causes runtime errors:

```
const user = {
  profile: {
    name: 'Alice',
    address: {
      city: 'Wonderland'
    }
  }
};

// Imperative risky access:
const city = user.profile.address.city; // safe here but often risky if any prop missing

// With Maybe Monad:
class Maybe {
  constructor(value) { this.value = value; }
  static of(value) { return new Maybe(value); }
  isNothing() { return this.value === null || this.value === undefined; }
  map(fn) {
    if (this.isNothing()) return Maybe.of(null);
    return Maybe.of(fn(this.value));
  }
  getOrElse(defaultValue) {
    return this.isNothing() ? defaultValue : this.value;
```

```
  }
}

const maybeCity = Maybe.of(user)
  .map(u => u.profile)
  .map(p => p.address)
  .map(a => a.city)
  .getOrElse('Unknown City');

console.log(maybeCity); // 'Wonderland'
```

If any property is missing, `Maybe` will safely propagate `Nothing`, avoiding exceptions, and return the default instead.

Handling Errors Explicitly with `Either`

`Either` allows distinguishing success from failure with useful error messages:

```
class Either {
  static right(value) { return new Right(value); }
  static left(value) { return new Left(value); }
}

class Right extends Either {
  constructor(value) { super(); this.value = value; }
  map(fn) { return Either.right(fn(this.value)); }
  chain(fn) { return fn(this.value); }
  fold(fLeft, fRight) { return fRight(this.value); }
}

class Left extends Either {
  constructor(value) { super(); this.value = value; }
  map(_) { return this; } // no-op on Left
  chain(_) { return this; }
  fold(fLeft, _) { return fLeft(this.value); }
}

// Example: safe JSON parsing
const safeJsonParse = str => {
  try {
    return Either.right(JSON.parse(str));
  } catch (e) {
    return Either.left('Invalid JSON');
  }
};

const jsonStr = '{"name": "Alice"}';

const result = safeJsonParse(jsonStr)
  .map(obj => obj.name)
  .fold(
    err => `Error: ${err}`,
    name => `Hello, ${name}!`
  );

console.log(result); // "Hello, Alice!"
```

If the JSON is invalid, the error is captured in `Left` and handled gracefully without throwing.

Summary: Safe Composition Benefits

- **Propagates absence or failure automatically** without manual error checks.
- **Avoids exceptions and null reference errors**, making code more robust.
- **Encourages declarative, composable pipelines** that are easier to test and reason about.
- **Separates normal flow from error handling**, improving code clarity.

By adopting `Maybe` and `Either`, you can build resilient, predictable functional programs that handle failure gracefully and keep your code clean and maintainable.

16.3 Practical Example: Avoiding Null Errors Functionally

One of the common challenges in JavaScript programming is dealing with **null** or **undefined** values, especially when accessing deeply nested properties. Without careful checks, these can cause runtime errors like `TypeError: Cannot read property 'x' of undefined`.

In this section, we'll use the `Maybe` monad to **eliminate explicit null checks** and write code that is safer, more declarative, and easier to maintain.

Imperative Approach with Null Checks

Consider this user data object, where some fields may be missing:

```
const user = {
  profile: {
    name: 'Alice',
    address: {
      city: 'Wonderland'
    }
  }
};

// Access city with nested null checks
function getCity(user) {
  if (user && user.profile && user.profile.address && user.profile.address.city) {
    return user.profile.address.city;
  }
  return 'Unknown City';
}

console.log(getCity(user));           // Wonderland
console.log(getCity({}));             // Unknown City
console.log(getCity(null));           // Unknown City
```

While this works, the explicit null checks quickly clutter the code and can be error-prone if you forget one.

Functional Approach Using the `Maybe` Monad

Let's rewrite this function using a simple `Maybe` monad abstraction to safely navigate nested properties without explicit checks.

```
class Maybe {
  constructor(value) {
    this.value = value;
  }

  static of(value) {
    return new Maybe(value);
  }

  isNothing() {
    return this.value === null || this.value === undefined;
  }

  map(fn) {
    if (this.isNothing()) return Maybe.of(null);
    return Maybe.of(fn(this.value));
  }

  getOrElse(defaultValue) {
    return this.isNothing() ? defaultValue : this.value;
  }
}

// Functional getCity using Maybe
function getCityFunctional(user) {
  return Maybe.of(user)
    .map(u => u.profile)
    .map(p => p.address)
    .map(a => a.city)
    .getOrElse('Unknown City');
}
const user = {
  profile: {
    name: 'Alice',
    address: {
      city: 'Wonderland'
    }
  }
};

console.log(getCityFunctional(user));           // Wonderland
console.log(getCityFunctional({}));             // Unknown City
console.log(getCityFunctional(null));           // Unknown City
```

Here's what happens:

- `Maybe.of(user)` wraps the input in a `Maybe`.
- `.map()` safely applies the function only if the value exists.
- If any intermediate step returns `null` or `undefined`, the chain short-circuits and results in `Nothing`.
- Finally, `.getOrElse()` provides a fallback default.

Benefits of the `Maybe` Approach

Imperative Style	Functional Style (Maybe)
Verbose nested `if` checks clutter logic	Clean, declarative chaining via `.map()` calls
Easy to forget null checks, causing runtime errors	Null safety baked into monad, avoiding errors
Harder to reason about flow and data transformations	Pipeline expresses intent clearly
Mixing business logic and error handling	Separation of concerns; fallback handled cleanly

Another Example: Optional Chaining Alternative

The modern JavaScript optional chaining (`?.`) operator solves some null check pains:

```
console.log(user?.profile?.address?.city ?? 'Unknown City'); // Wonderland
```

However, the `Maybe` monad approach extends beyond just property access — it integrates naturally with other functional operations like `map`, `chain`, and composition, offering a unified paradigm for safer, more predictable code.

Summary

Using the `Maybe` monad to handle nullable or optional data:

- **Eliminates repetitive null checks**,
- **Improves readability and maintainability**,
- **Makes the codebase more robust against runtime errors**,
- **Fits naturally in a functional programming style**, enabling easy composition and testing.

This technique helps you write JavaScript that is safe, elegant, and easy to extend — a hallmark of effective functional programming.

Chapter 17.

Functional Reactive Programming (FRP)

1. What Is FRP?
2. Streams and Observables
3. Using RxJS for Functional Event Handling

17 Functional Reactive Programming (FRP)

17.1 What Is FRP?

Functional Reactive Programming (FRP) is a programming paradigm that combines **functional programming** with **reactive programming** concepts to handle asynchronous data streams in a declarative and composable way.

At its core, FRP treats **time-varying values** and **events** as streams that can be manipulated using functional operators. Instead of imperatively managing event listeners and callbacks, FRP encourages you to think of data as continuous flows that react to changes over time.

Core Ideas of FRP

- **Declarative Event Handling**: You describe *what* should happen when events occur, rather than *how* to handle each event imperatively.
- **Continuous Data Flows**: Values are modeled as streams that change over time, such as mouse positions, keyboard input, or server responses.
- **Functional Operators**: You can transform, filter, combine, and compose these streams using pure functions, similar to how you work with arrays.

How FRP Differs from Traditional Imperative Event Handling

In traditional imperative JavaScript, handling user input or asynchronous data often involves:

- Explicitly adding event listeners (e.g., `element.addEventListener`),
- Writing callback functions,
- Managing state and side effects manually,
- Dealing with nested callbacks or complex state machines.

FRP abstracts this by treating events as streams you can *compose* and *transform*. This leads to clearer, more maintainable code by focusing on **what data flows to listen to** and **how to transform those flows**, rather than on the mechanics of event registration.

Conceptual Example: Mouse Movements as a Stream

Imagine tracking mouse movements on a page:

- **Imperative approach**: You attach a listener and update state or UI inside the callback.

```
document.addEventListener('mousemove', event => {
  console.log(`Mouse at (${event.clientX}, ${event.clientY})`);
});
```

- **FRP approach**: You model mouse movements as a stream of events and apply transformations declaratively.

```
const mouseMoves = fromEvent(document, 'mousemove'); // Stream of mouse events

const positions = mouseMoves.map(event => ({
  x: event.clientX,
  y: event.clientY,
}));

positions.subscribe(pos => console.log(`Mouse at (${pos.x}, ${pos.y})`));
```

Here, `fromEvent` creates an **observable stream** of mouse events, and `.map()` transforms the stream data. You then **subscribe** to the stream to react to new positions.

Benefits of FRP

- **Simplified asynchronous handling**: Complex event sequences become easier to manage.
- **Improved readability**: Declarative code expresses *what* happens rather than *how*.
- **Composable logic**: Stream operators let you build complex behaviors from simple building blocks.
- **Better scalability**: Easier to reason about events and side effects, especially in real-time or interactive apps.

In summary, Functional Reactive Programming lets you work with asynchronous events and time-varying data as **pure, composable streams**, making your code more declarative, modular, and easier to reason about. It's an invaluable toolset for building responsive, real-time JavaScript applications.

17.2 Streams and Observables

In Functional Reactive Programming (FRP), the two fundamental abstractions you'll work with are **streams** and **observables**. Understanding these is key to handling asynchronous events and data flows in a declarative, functional way.

What Are Streams?

A **stream** represents a sequence of events or values that occur over time. These can be anything from:

- User inputs (clicks, key presses, mouse movements),
- Data from a server (HTTP responses, WebSocket messages),
- Timers and intervals,
- Or any other asynchronous data source.

Streams are *continuous* and *ordered* — they emit values in a defined sequence, much like an array, but spread out over time rather than all at once.

What Are Observables?

An **observable** is a producer of streams. It's an object or entity that emits values to its subscribers over time.

- Observables can emit **zero or more** values,
- They can emit **events asynchronously**,
- They support **subscription** so observers can react to these events,
- And they allow **transformation** of the emitted values via operators.

Creating and Using Observables

Let's look at an example using RxJS, a popular FRP library that implements observables:

```
import { fromEvent } from 'rxjs';
import { map, filter, debounceTime } from 'rxjs/operators';

// Create an observable from click events on a button
const clicks$ = fromEvent(document.querySelector('button'), 'click');

// Transform the stream: map event to coordinates, filter, and debounce
const processedClicks$ = clicks$.pipe(
  map(event => ({ x: event.clientX, y: event.clientY })),
  filter(({ x, y }) => x > 100),          // Only clicks beyond x=100
  debounceTime(300)                        // Wait 300ms before emitting
);

// Subscribe to listen and react to the transformed stream
processedClicks$.subscribe(pos => {
  console.log(`Processed click at (${pos.x}, ${pos.y})`);
});
```

- `fromEvent()` creates an observable stream of click events.
- `.pipe()` chains operators to transform the stream.
- `map()` projects events into a simpler form.
- `filter()` selectively passes events meeting a condition.
- `debounceTime()` throttles the events, emitting only if no new event occurs within 300ms.
- `.subscribe()` attaches a listener to receive and act on emitted values.

Visualizing Streams and Observables

```
Observable (Producer)
     |
     v
 Stream of Events ---> [map()] ---> [filter()] ---> [debounceTime()] ---> Subscriber (Ob
```

- The observable emits raw events.
- Each operator transforms or filters the events as they flow downstream.
- Finally, subscribers consume the processed events.

Summary

- **Streams** represent asynchronous sequences of data over time.
- **Observables** are the producers that emit stream events.
- Using functional operators like `map`, `filter`, and `debounce`, you can declaratively transform streams.
- Subscribers listen to and react to these transformed data flows.

Together, streams and observables provide a powerful foundation for building responsive, event-driven applications in a clean, composable way.

17.3 Using RxJS for Functional Event Handling

RxJS (Reactive Extensions for JavaScript) is a powerful library that brings Functional Reactive Programming (FRP) concepts to JavaScript. It enables you to work with asynchronous data streams in a declarative, composable, and functional style.

Overview of RxJS

- **RxJS** provides a rich set of tools for creating, transforming, and combining **observables**.
- It supports **automatic subscription management** and powerful **operators** for handling events.
- RxJS encourages writing **pure, declarative code** that describes *what* to do with streams rather than *how* to do it imperatively.
- This results in **cleaner, more maintainable, and testable asynchronous code** for interactive applications.

Basic RxJS Usage

Let's look at the core building blocks:

- **Creating Observables:** You can create streams from various sources such as DOM events, promises, or intervals.
- **Subscribing:** Observers subscribe to streams to react to emitted events.
- **Operators:** Functions like `map`, `filter`, `debounceTime`, and `distinctUntilChanged` allow you to transform and control streams.

Example: Debounced Search Box

This example shows how to build a search input box that waits for the user to stop typing before firing a search, reducing unnecessary calls.

```
<input type="text" id="search" placeholder="Type to search..." />
```

```
import { fromEvent } from 'rxjs';
import { debounceTime, map, distinctUntilChanged } from 'rxjs/operators';

const searchBox = document.getElementById('search');

// Create observable from input events
const input$ = fromEvent(searchBox, 'input');

input$.pipe(
  map(event => event.target.value.trim()),    // Extract input value
  debounceTime(300),                          // Wait 300ms pause in typing
  distinctUntilChanged()                      // Only proceed if value changed
).subscribe(searchTerm => {
  console.log('Searching for:', searchTerm);
  // Here you could trigger an API call or update UI
});
```

What happens here?

- `fromEvent` creates a stream of input events.
- The pipeline:
  - Extracts the trimmed input value,
  - Waits until typing pauses for 300ms,
  - Ignores repeated inputs with the same value,
- Finally, the subscription receives the debounced search term.

Example: Live Mouse Position Tracker

Here's another interactive example showing how RxJS can be used to track mouse movements:

```
import { fromEvent } from 'rxjs';
import { map, throttleTime } from 'rxjs/operators';

const mouseMoves$ = fromEvent(document, 'mousemove');

mouseMoves$.pipe(
  throttleTime(100),  // Limit event frequency to one every 100ms
  map(event => ({ x: event.clientX, y: event.clientY }))
).subscribe(pos => {
  console.log(`Mouse at (${pos.x}, ${pos.y})`);
  // You could update the UI or perform other actions here
});
```

Why RxJS Embraces Functional Principles

- **Immutability:** RxJS operators do not mutate the original streams; they return new streams.
- **Pure functions:** Operators like `map`, `filter`, and `reduce` transform streams without side effects.
- **Declarative:** You describe your event logic as a series of transformations, not imperative event handling.

- **Composability:** Small, reusable operators can be combined to build complex asynchronous behaviors.

This approach leads to asynchronous code that is easier to reason about, debug, and test — essential qualities for modern reactive applications.

RxJS is an essential tool for developers looking to apply functional programming to real-world event-driven JavaScript applications. Its elegant handling of streams makes managing complex asynchronous flows intuitive and maintainable.

Chapter 18.

Functional State Management

1. Pure Reducers and State Transitions
2. Building a Mini Redux-Style Store
3. Practical Example: Functional Todo App Logic

18 Functional State Management

18.1 Pure Reducers and State Transitions

In functional programming, managing state changes predictably and safely is crucial. **Pure reducers** play a central role in this process by defining how state evolves in response to actions.

What Is a Pure Reducer?

A **reducer** is a function that:

- Takes two inputs:
  - The **current state**
  - An **action** describing what happened
- Returns a **new state** based solely on these inputs

Importantly, **pure reducers do not cause side effects** — they do not modify the current state directly, perform I/O, or depend on external variables. Instead, they produce a fresh, updated state object, leaving the original untouched.

This purity guarantees that given the same state and action, a reducer will always return the same new state. This predictability is foundational for reliable, testable state management.

State Immutability and Predictable Transitions

Reducers enforce **immutability** by never mutating the existing state object. Instead, they create and return new state copies with the necessary updates. This approach:

- Avoids bugs caused by accidental state mutation
- Enables time-travel debugging and undo functionality
- Simplifies reasoning about state changes over time

Writing Simple Reducers: Examples

Here are examples of reducers managing simple state shapes with common actions:

Counter Reducer

```
function counterReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    case 'DECREMENT':
      return { ...state, count: state.count - 1 };
    case 'RESET':
      return { ...state, count: 0 };
    default:
      return state;
  }
}
```

- Takes current state `{ count }` and an action `{ type }`
- Returns new state with updated count, using spread syntax to preserve immutability
- Returns original state if action type is unrecognized

Todo List Reducer

```
function todosReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state, { id: action.id, text: action.text, completed: false }];
    case 'TOGGLE_TODO':
      return state.map(todo =>
        todo.id === action.id
          ? { ...todo, completed: !todo.completed }
          : todo
      );
    case 'REMOVE_TODO':
      return state.filter(todo => todo.id !== action.id);
    default:
      return state;
  }
}
```

- Manages an array of todo items immutably
- Uses `map` and `filter` to produce new arrays rather than modifying existing ones

Why Purity Matters in State Management

- **Testability:** Pure reducers are easy to test since they're deterministic.
- **Debugging:** You can reproduce any state by replaying actions.
- **Modularity:** Reducers can be composed or split for maintainability.
- **Concurrency:** Immutable state prevents race conditions in async updates.

By embracing pure reducers and immutable state transitions, you lay the foundation for robust, predictable applications — making functional state management both powerful and intuitive.

18.2 Building a Mini Redux-Style Store

In this section, we'll build a minimal **Redux-style store** from scratch to manage application state functionally. This store will serve as a **state container**, allow **dispatching actions**, and support **subscribing to state changes** — all while adhering to immutability and pure reducer principles.

Core Concepts

- **State Container:** Holds the current application state internally.
- **Dispatch:** A method to send actions to the store, triggering state updates.
- **Subscribe:** Allows registering listeners to be notified whenever state changes.

- **Reducer:** A pure function that determines how state updates based on actions.

Step 1: Define the Store Factory Function

The store will be created with a reducer function and initial state:

```
function createStore(reducer, initialState) {
  let state = initialState;
  let listeners = [];

  // Returns the current state (read-only)
  function getState() {
    return state;
  }

  // Dispatches an action to update state via reducer
  function dispatch(action) {
    // Call reducer with current state and action
    const newState = reducer(state, action);

    // Only update if state has changed
    if (newState !== state) {
      state = newState;

      // Notify all subscribers
      listeners.forEach(listener => listener());
    }
  }

  // Registers a listener callback, returns unsubscribe function
  function subscribe(listener) {
    listeners.push(listener);

    // Return an unsubscribe function
    return () => {
      listeners = listeners.filter(l => l !== listener);
    };
  }

  // Initialize state by dispatching an empty action
  dispatch({ type: '@@INIT' });

  return { getState, dispatch, subscribe };
}
```

Explanation

- **`state`** holds the current state, private inside the closure.
- **`listeners`** is an array of callback functions to notify on changes.
- **`getState()`** returns the current state, ensuring read-only access.
- **`dispatch(action)`**:
  - Calls the reducer with current state and action.
  - If state changes (reference inequality), updates **`state`** and calls all listeners.

- **`subscribe(listener)`**:
  - Adds a listener to `listeners`.
  - Returns an unsubscribe function to remove it later.
- The store initializes its state by dispatching a dummy action.

Step 2: Usage Example with a Reducer

Let's create a simple counter reducer and use the store:

```
// Reducer: manages state updates based on action.type
function counterReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

// Create store with reducer and initial state
const store = createStore(counterReducer);

// Subscribe to state changes
const unsubscribe = store.subscribe(() => {
  console.log('State changed:', store.getState());
});

// Dispatch some actions
store.dispatch({ type: 'INCREMENT' }); // State changed: { count: 1 }
store.dispatch({ type: 'INCREMENT' }); // State changed: { count: 2 }
store.dispatch({ type: 'DECREMENT' }); // State changed: { count: 1 }

// Stop listening to state changes
unsubscribe();
```

Full runnable code:

```
// Step 1: Store factory function
function createStore(reducer, initialState) {
  let state = initialState;
  let listeners = [];

  function getState() {
    return state;
  }

  function dispatch(action) {
    const newState = reducer(state, action);
    if (newState !== state) {
      state = newState;
      listeners.forEach(listener => listener());
    }
```

```
  }

  function subscribe(listener) {
    listeners.push(listener);
    return () => {
      listeners = listeners.filter(l => l !== listener);
    };
  }

  dispatch({ type: '@@INIT' }); // Initialize state

  return { getState, dispatch, subscribe };
}

// Step 2: Reducer function
function counterReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

// Create the store
const store = createStore(counterReducer);

// Subscribe to state changes
const unsubscribe = store.subscribe(() => {
  console.log('State changed:', store.getState());
});

// Dispatch actions
store.dispatch({ type: 'INCREMENT' }); // -> { count: 1 }
store.dispatch({ type: 'INCREMENT' }); // -> { count: 2 }
store.dispatch({ type: 'DECREMENT' }); // -> { count: 1 }

// Stop listening
unsubscribe();

// Further dispatch (no log output expected)
store.dispatch({ type: 'INCREMENT' }); // -> no output
```

Key Points

- The store **maintains immutable state** by delegating updates to the pure reducer.
- State changes are **observable** via subscriptions.
- Dispatching an action triggers state transition and notifies subscribers.
- The store's internal state is **encapsulated** and only accessible via `getState()`.

Why Build Your Own Store?

- Reinforces understanding of functional state management concepts.
- Helps appreciate Redux's simplicity and power.

- Enables customizing behavior for specialized scenarios.

By building this minimal Redux-style store, you gain hands-on experience with functional programming principles applied to state management — immutability, pure reducers, and declarative updates. This foundation prepares you to use or extend mature libraries effectively.

18.3 Practical Example: Functional Todo App Logic

In this section, we'll develop a complete, self-contained example of the core state logic for a **todo app** using functional programming principles. We'll focus on **pure reducers** that handle adding, toggling, and deleting todos immutably, and show how actions flow through the store to update the state predictably.

Step 1: Define the Todo Actions

First, we define the action types and action creators:

```
// Action Types
const ADD_TODO = 'ADD_TODO';
const TOGGLE_TODO = 'TOGGLE_TODO';
const DELETE_TODO = 'DELETE_TODO';

// Action Creators
const addTodo = (text) => ({ type: ADD_TODO, payload: { text } });
const toggleTodo = (id) => ({ type: TOGGLE_TODO, payload: { id } });
const deleteTodo = (id) => ({ type: DELETE_TODO, payload: { id } });
```

Step 2: Write the Reducer

The reducer manages the state, which is an array of todo objects. Each todo has an `id`, `text`, and a `completed` flag.

```
// Initial state: empty todo list
const initialState = [];

// Pure reducer function
function todosReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO: {
      const newTodo = {
        id: Date.now(),  // unique ID using timestamp
        text: action.payload.text,
        completed: false,
      };
      // Return a new array with the new todo appended
      return [...state, newTodo];
    }

    case TOGGLE_TODO: {
      // Return a new array with toggled 'completed' for matching todo
```

```
      return state.map(todo =>
        todo.id === action.payload.id
          ? { ...todo, completed: !todo.completed }
          : todo
      );
    }

    case DELETE_TODO: {
      // Return a new array filtering out the todo with matching id
      return state.filter(todo => todo.id !== action.payload.id);
    }

    default:
      return state;  // Return current state for unknown actions
  }
}
```

Step 3: Integrate With the Store

Using the `createStore` function from the previous section, we create the store with this reducer:

```
const store = createStore(todosReducer);
```

Step 4: Dispatch Actions and Observe State Changes

```
// Subscribe to state changes for debugging
const unsubscribe = store.subscribe(() => {
  console.log('Current Todos:', store.getState());
});

// Add some todos
store.dispatch(addTodo('Learn Functional Programming'));
store.dispatch(addTodo('Write a blog post'));

// Toggle the first todo (simulate by accessing the first todo's id)
const firstTodoId = store.getState()[0].id;
store.dispatch(toggleTodo(firstTodoId));

// Delete the second todo
const secondTodoId = store.getState()[1].id;
store.dispatch(deleteTodo(secondTodoId));

// Stop listening when done
unsubscribe();
```

Output Sample:

```
Current Todos: [{ id: 1679900000000, text: 'Learn Functional Programming', completed: fa
Current Todos: [
  { id: 1679900000000, text: 'Learn Functional Programming', completed: false },
  { id: 1679900001000, text: 'Write a blog post', completed: false }
```

```
]
Current Todos: [
  { id: 1679900000000, text: 'Learn Functional Programming', completed: true },
  { id: 1679900001000, text: 'Write a blog post', completed: false }
]
Current Todos: [
  { id: 1679900000000, text: 'Learn Functional Programming', completed: true }
]
```

Full runnable code:

```
// ----------------------------
// Step 1: Action Types & Creators
// ----------------------------
const ADD_TODO = 'ADD_TODO';
const TOGGLE_TODO = 'TOGGLE_TODO';
const DELETE_TODO = 'DELETE_TODO';

const addTodo = (text) => ({ type: ADD_TODO, payload: { text } });
const toggleTodo = (id) => ({ type: TOGGLE_TODO, payload: { id } });
const deleteTodo = (id) => ({ type: DELETE_TODO, payload: { id } });

// ----------------------------
// Step 2: Reducer Function
// ----------------------------
const initialState = [];

function todosReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          id: Date.now(),
          text: action.payload.text,
          completed: false,
        },
      ];

    case TOGGLE_TODO:
      return state.map(todo =>
        todo.id === action.payload.id
          ? { ...todo, completed: !todo.completed }
          : todo
      );

    case DELETE_TODO:
      return state.filter(todo => todo.id !== action.payload.id);

    default:
      return state;
  }
}

// ----------------------------
// Step 3: Store Implementation
```

```
// ----------------------------
function createStore(reducer, initialState) {
  let state = initialState;
  let listeners = [];

  function getState() {
    return state;
  }

  function dispatch(action) {
    const newState = reducer(state, action);
    if (newState !== state) {
      state = newState;
      listeners.forEach(listener => listener());
    }
  }

  function subscribe(listener) {
    listeners.push(listener);
    return () => {
      listeners = listeners.filter(l => l !== listener);
    };
  }

  dispatch({ type: '@@INIT' });
  return { getState, dispatch, subscribe };
}

// ----------------------------
// Step 4: Using the Store
// ----------------------------
const store = createStore(todosReducer);

// Subscribe to log state on change
const unsubscribe = store.subscribe(() => {
  console.log('Current Todos:', store.getState());
});

// Add todos
store.dispatch(addTodo('Learn Functional Programming'));
store.dispatch(addTodo('Write a blog post'));

// Toggle first todo
const firstTodoId = store.getState()[0].id;
store.dispatch(toggleTodo(firstTodoId));

// Delete second todo
const secondTodoId = store.getState()[1].id;
store.dispatch(deleteTodo(secondTodoId));

// Unsubscribe
unsubscribe();
```

Step 5: Testing and Debugging Tips

- **Test Reducer Independently:** Since reducers are pure functions, you can write unit tests passing known state and actions, asserting expected new state.

- **Log State Changes:** Subscribe to the store and log state after each dispatch for real-time debugging.
- **Use Immutable Updates:** Always return new state objects without mutating existing arrays or objects to prevent bugs.
- **Action Creators Help:** Use action creators to keep action construction consistent and easier to maintain.

Why Functional Patterns Simplify State Management

- **Predictable State Transitions:** Pure reducers guarantee the same output for given inputs, making app behavior easier to understand and debug.
- **Immutability:** Creating new state objects prevents unintended side effects and allows time-travel debugging or undo functionality.
- **Separation of Concerns:** Reducers focus only on how state changes, independent of UI rendering or side effects.
- **Testability:** Pure functions are easy to test in isolation without needing complex setups.

This functional approach provides a robust foundation for building complex UI logic in a maintainable and scalable way, enabling you to confidently manage state and evolve your app without tangled mutable code.

Chapter 19.

Functional Programming in the Browser

1. Functional DOM Event Handling
2. Pure View Rendering with Virtual DOM Concepts
3. Real Example: Functional Click Tracker

19 Functional Programming in the Browser

19.1 Functional DOM Event Handling

Handling DOM events in the browser is a crucial part of building interactive web applications. Applying **functional programming (FP) principles** to this task can lead to cleaner, more predictable, and reusable code. This section explains how to write **pure event handlers**, avoid side effects, and leverage techniques like **event delegation**, **higher-order functions**, and **function composition** for functional DOM event handling.

Avoiding Side Effects in Event Handlers

In imperative programming, event handlers often mutate external state or manipulate the DOM directly, which can lead to unpredictable behavior and harder-to-maintain code.

Functional programming encourages writing event handlers as pure functions:

- They **do not modify external state** directly.
- They **delegate state changes** to pure functions or centralized state managers.
- They **return new data or trigger effects indirectly** instead of performing mutations inline.

Example: Impure vs Pure Event Handler

```
// Impure event handler: directly mutates external state
let counter = 0;
button.addEventListener('click', () => {
  counter++;            // Side effect: modifying external state
  console.log(counter);
});

// Pure event handler: delegates state change
function onClick(currentCounter) {
  return currentCounter + 1;  // Returns new state, no side effects
}

// Usage example
let counter = 0;
button.addEventListener('click', () => {
  counter = onClick(counter);
  console.log(counter);
});
```

Notice how the pure version delegates the update logic to `onClick` and only reassigns state outside the pure function, making the core logic easier to test and reason about.

Using Higher-Order Functions for Reusable Handlers

Higher-order functions—functions that accept or return other functions—allow you to create flexible and reusable event handlers.

```
// A higher-order function that creates a click handler with custom logic
function createClickHandler(updateFn) {
  return function(event) {
    // Pure handler delegates to update function
    const newState = updateFn(event);
    console.log('New state:', newState);
  };
}

// Example usage with a simple updater function
const incrementOnClick = createClickHandler(() => counter + 1);

let counter = 0;
button.addEventListener('click', (e) => {
  counter = incrementOnClick(e);
});
```

By separating **event handling logic** from **state updates**, this pattern encourages composition and code reuse.

Event Delegation with Functional Handlers

Event delegation improves performance and simplifies event management by attaching a single event listener to a parent element rather than many child elements.

A functional event handler can use event delegation while remaining pure:

```
const list = document.querySelector('#todo-list');

function handleItemClick(event) {
  const item = event.target.closest('.todo-item');
  if (!item) return; // Ignore clicks outside todo items
  // Delegate action, e.g., toggle completion
  toggleTodoItem(item.dataset.id);
}

// Attach a single delegated event handler
list.addEventListener('click', handleItemClick);
```

Here, `handleItemClick` is a pure function that does not mutate state directly but delegates to `toggleTodoItem`, which should itself be a pure function returning updated state.

Composing Event Handlers

You can use function composition to build complex handlers from smaller, reusable pieces:

```
// Small utility to log events
const logEvent = (fn) => (event) => {
  console.log('Event:', event.type);
  return fn(event);
};

// Small utility to prevent default behavior
const preventDefault = (fn) => (event) => {
  event.preventDefault();
```

```
  return fn(event);
};

// Compose a handler that prevents default and logs event
const composedHandler = (event) => {
  console.log('Clicked!');
};

const enhancedHandler = preventDefault(logEvent(composedHandler));

button.addEventListener('click', enhancedHandler);
```

This approach cleanly layers concerns without duplicating logic or mixing side effects.

19.1.1 Summary

Functional DOM event handling emphasizes:

- Writing **pure event handlers** that avoid side effects and delegate state changes.
- Using **higher-order functions** to create reusable, composable handlers.
- Applying **event delegation** functionally to efficiently manage events.
- Leveraging **function composition** to build complex behaviors from simple utilities.

By adopting these FP techniques, your event handling code becomes more modular, predictable, and easier to test and maintain — all key goals in modern front-end development.

19.2 Pure View Rendering with Virtual DOM Concepts

19.2.1 Pure View Rendering with Virtual DOM Concepts

Functional programming principles greatly influence modern UI development, especially in how views (the user interface) are rendered and updated. A core idea is **pure view rendering**—writing functions that take application state as input and return a description of the UI without causing side effects or directly manipulating the DOM.

What Is Pure View Rendering?

A **pure rendering function**:

- Takes the current **state** as input.
- Returns a **UI representation** (often a virtual DOM tree or similar structure).
- Does **not** perform side effects like modifying the actual DOM, triggering animations, or making API calls.
- Is **deterministic**: given the same state, it always produces the same UI.

This makes UI logic predictable, testable, and easy to reason about—just like pure functions in functional programming.

Introducing the Virtual DOM

The **Virtual DOM (VDOM)** is an abstraction used by many modern frameworks (like React) that represents the UI as a lightweight JavaScript object tree rather than directly manipulating browser DOM nodes.

How it works:

1. **Render:** You write a pure rendering function that converts your app's state into a virtual DOM tree.
2. **Diff:** When the state changes, you generate a new virtual DOM tree and compare ("diff") it with the previous one.
3. **Patch:** Only the differences between the old and new trees are applied to the real DOM, minimizing costly updates.

This approach enables:

- **Declarative UI updates:** You describe *what* the UI should look like for any state, not *how* to mutate the DOM step-by-step.
- **Immutability:** Each state produces a fresh virtual tree without mutating previous versions.
- **Performance:** Efficient updates by batching and minimizing direct DOM operations.

Simple Example of a Pure Rendering Function

```
// Pure function returning a virtual DOM-like object representing UI for a counter
function renderCounter(state) {
  return {
    tag: 'div',
    children: [
      { tag: 'h1', children: [`Count: ${state.count}`] },
      { tag: 'button', props: { id: 'inc' }, children: ['Increment'] },
      { tag: 'button', props: { id: 'dec' }, children: ['Decrement'] },
    ],
  };
}

// Example state
const state = { count: 5 };
const vdom = renderCounter(state);
console.log(vdom);
```

This function is:

- **Pure:** It only depends on `state`.
- **Declarative:** It describes the UI as a tree of objects.
- **Side-effect free:** It doesn't manipulate the DOM directly.

Virtual DOM Diffing Concept (Pseudocode)

When state updates, the framework:

1. Calls `renderCounter(newState)` → new virtual DOM tree.
2. Compares `newVDOM` with `oldVDOM`.
3. Identifies changes (e.g., text changed, nodes added/removed).
4. Applies minimal DOM operations to update the real UI.

```
function diff(oldNode, newNode) {
  if (!oldNode) {
    createElement(newNode);  // Insert new element
  } else if (!newNode) {
    removeElement(oldNode);  // Remove old element
  } else if (changed(oldNode, newNode)) {
    replaceElement(oldNode, newNode);  // Replace if node type or content changed
  } else {
    // Recursively diff children
    for (let i = 0; i < Math.max(oldNode.children.length, newNode.children.length); i++) {
      diff(oldNode.children[i], newNode.children[i]);
    }
  }
}
```

Benefits of Pure View Rendering with Virtual DOM

- **Predictability:** UI is a pure function of state, making debugging straightforward.
- **Performance:** Virtual DOM optimizes updates, reducing expensive DOM manipulations.
- **Modularity:** Components are composable pure functions.
- **Testability:** Pure rendering functions can be tested without a browser environment.
- **Immutability:** Encourages treating state as immutable, simplifying change detection.

19.2.2 Summary

Pure view rendering with virtual DOM is a powerful pattern that aligns perfectly with functional programming ideas:

- Treat UI as data, not as mutable DOM instructions.
- Write pure functions that map state to UI descriptions.
- Use virtual DOM diffing to efficiently update the real UI.

This approach leads to clearer, maintainable, and performant UI code—fundamental for modern web applications.

19.3 Real Example: Functional Click Tracker

Let's build a fully functional **click tracker** app in the browser using functional programming principles. This example will demonstrate:

- Using **pure functions** to update state
- Declarative rendering of UI from state
- Avoiding direct DOM mutations inside event handlers
- Clear separation of concerns for maintainability and testability

Architecture Overview

1. **State**: A simple object holding the click count.
2. **Reducer (Pure function)**: Takes current state and an action, returns new state.
3. **Render function (Pure function)**: Takes state and returns a virtual DOM representation or directly updates the DOM declaratively.
4. **Event Handler**: Dispatches actions to update state without mutating it.
5. **Render cycle**: After each state update, the UI is re-rendered from scratch based on new state.

Complete Code Example

Style:

```
<style>
  body { font-family: sans-serif; padding: 2rem; }
  button { font-size: 1.2rem; padding: 0.5rem 1rem; }
</style>
```

Html:

```
<div id="app"></div>
```

Javascript:

```
// Initial state
const initialState = { count: 0 };

// Pure reducer function: returns new state based on action
function clickReducer(state, action) {
  switch(action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    default:
      return state;
  }
}

// Pure render function: creates UI from state
function render(state) {
  return `
    <h1>Click Tracker</h1>
```

```
        <p>Button clicked <strong>${state.count}</strong> times</p>
        <button id="incrementBtn">Click me</button>
      `;
    }

    // Store to hold current state and render UI on changes
    function createStore(reducer, initialState) {
      let state = initialState;
      const listeners = [];

      // Get current state
      const getState = () => state;

      // Dispatch action to update state
      const dispatch = (action) => {
        const newState = reducer(state, action);
        if (newState !== state) {
          state = newState;
          listeners.forEach(listener => listener());
        }
      };

      // Subscribe to state changes
      const subscribe = (listener) => {
        listeners.push(listener);
        return () => {
          const index = listeners.indexOf(listener);
          if (index > -1) listeners.splice(index, 1);
        };
      };

      return { getState, dispatch, subscribe };
    }

    // Initialize store
    const store = createStore(clickReducer, initialState);

    // Main render loop
    function renderApp() {
      const app = document.getElementById('app');
      app.innerHTML = render(store.getState());

      // Attach event handler declaratively without side effects on state
      const button = document.getElementById('incrementBtn');
      button.addEventListener('click', () => {
        store.dispatch({ type: 'INCREMENT' });
      });
    }

    // Subscribe render to state changes
    store.subscribe(renderApp);

    // Initial render
    renderApp();
```

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

```
<head>
  <meta charset="UTF-8" />
  <title>Functional Click Tracker</title>
  <style>
    body { font-family: sans-serif; padding: 2rem; }
    button { font-size: 1.2rem; padding: 0.5rem 1rem; }
  </style>
</head>
<body>
  <div id="app"></div>

  <script>
    // Initial state
    const initialState = { count: 0 };

    // Pure reducer function: returns new state based on action
    function clickReducer(state, action) {
      switch(action.type) {
        case 'INCREMENT':
          return { count: state.count + 1 };
        default:
          return state;
      }
    }

    // Pure render function: creates UI from state
    function render(state) {
      return `
        <h1>Click Tracker</h1>
        <p>Button clicked <strong>${state.count}</strong> times</p>
        <button id="incrementBtn">Click me</button>
      `;
    }

    // Store to hold current state and render UI on changes
    function createStore(reducer, initialState) {
      let state = initialState;
      const listeners = [];

      // Get current state
      const getState = () => state;

      // Dispatch action to update state
      const dispatch = (action) => {
        const newState = reducer(state, action);
        if (newState !== state) {
          state = newState;
          listeners.forEach(listener => listener());
        }
      };

      // Subscribe to state changes
      const subscribe = (listener) => {
        listeners.push(listener);
        return () => {
          const index = listeners.indexOf(listener);
          if (index > -1) listeners.splice(index, 1);
        };
```

```
    };

    return { getState, dispatch, subscribe };
  }

  // Initialize store
  const store = createStore(clickReducer, initialState);

  // Main render loop
  function renderApp() {
    const app = document.getElementById('app');
    app.innerHTML = render(store.getState());

    // Attach event handler declaratively without side effects on state
    const button = document.getElementById('incrementBtn');
    button.addEventListener('click', () => {
      store.dispatch({ type: 'INCREMENT' });
    });
  }

  // Subscribe render to state changes
  store.subscribe(renderApp);

  // Initial render
  renderApp();
 </script>
</body>
</html>
```

Explanation

- The **`clickReducer`** is a pure function: it only depends on its inputs (`state`, `action`) and returns a new state without side effects.
- The **`render`** function takes the current state and returns a string of HTML representing the UI. It doesn't manipulate the DOM directly.
- The **store** encapsulates the state and provides methods to dispatch actions and subscribe to state changes, keeping the state immutable and centralized.
- On each dispatch, the store updates the state immutably and notifies subscribers.
- The **`renderApp`** function reacts to state changes by re-rendering the UI completely, then attaching event handlers.
- Event handlers **do not mutate state** directly; they dispatch actions to the store.

Why Functional?

- **Predictable state transitions:** State is updated only through pure reducer functions.
- **No direct DOM mutations in handlers:** UI rendering is decoupled from event handling.
- **Immutability:** The state is replaced, not mutated, easing debugging and time-traveling (if desired).
- **Testability:** Reducers and render functions can be tested independently of DOM or events.
- **Maintainability:** Clear separation of state management, rendering, and event logic.

This simple click tracker illustrates how functional programming concepts can build clean, maintainable browser apps with clear flow and minimal side effects. As apps grow, this architecture scales well with more complex state and UI logic.

Chapter 20.

Functional Programming in Node.js

1. Using Functional Patterns in Server Logic
2. Stream and File Processing
3. CLI Tools with Pure Functions

20 Functional Programming in Node.js

20.1 Using Functional Patterns in Server Logic

Functional programming principles can greatly improve the design, readability, and maintainability of Node.js server-side applications. By favoring **pure functions**, **immutable data**, and **function composition**, you can build modular, predictable, and testable server logic.

Pure Functions in Server Logic

At the heart of functional programming are **pure functions** — functions that:

- Always return the same output for the same input
- Have no side effects (e.g., no modifying external state or I/O directly)

In a Node.js server, you can write **pure validation functions**, **business logic**, and even parts of **request handlers** as pure functions.

Example: Pure Validation Function

```
// Pure function to validate user input
function validateUser(data) {
  const errors = [];
  if (!data.username || data.username.length < 3) {
    errors.push('Username must be at least 3 characters.');
  }
  if (!data.email || !data.email.includes('@')) {
    errors.push('Invalid email address.');
  }
  return errors.length ? errors : null;
}
```

This function is deterministic and side-effect free, making it easy to test and reuse.

Composable Middleware Chains

Node.js servers often use middleware functions to process requests. By writing middleware as **pure, composable functions**, you can build flexible pipelines that are easy to reason about.

Example: Simple Composable Middleware

```
const express = require('express');
const app = express();

// Middleware factory: logs request method and URL
const logger = () => (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
};

// Middleware factory: checks if request has a valid API key
const checkApiKey = (validKey) => (req, res, next) => {
```

```
  if (req.headers['api-key'] === validKey) {
    next();
  } else {
    res.status(403).send('Forbidden');
  }
};

// Compose middlewares
app.use(logger());
app.use(checkApiKey('secret123'));

app.get('/', (req, res) => {
  res.send('Hello, functional world!');
});

app.listen(3000);
```

Here, `logger` and `checkApiKey` are higher-order functions producing middleware that can be composed and reused.

Immutable Data Flow

Avoid mutating request or response objects directly. Instead, create new objects representing updated state or data.

Example: Immutable Update in Business Logic

```
function updateUserProfile(user, updates) {
  // Return new user object without mutating original
  return { ...user, ...updates };
}
```

Immutable data flow prevents subtle bugs and helps ensure consistency across asynchronous server operations.

Benefits for Scalability and Maintainability

- **Modularity**: Pure functions and composable middleware can be developed and tested in isolation.
- **Testability**: With no side effects, testing is simpler and less brittle.
- **Predictability**: Pure functions always produce the same output for the same input, making bugs easier to locate.
- **Ease of Reasoning**: Function composition encourages declarative and readable server logic.
- **Scalability**: Modular components can be extended or replaced without impacting unrelated parts.

Applying functional programming in Node.js backend development leads to cleaner, more reliable, and maintainable server applications. The principles you learn here will serve well as your applications grow in complexity and scale.

20.2 Stream and File Processing

Node.js streams are powerful abstractions for processing data incrementally, such as reading or writing files, network communications, or any data flow. Handling streams **functionally** means composing transformations as pure functions, avoiding side effects, and leveraging built-in pipeline utilities for clean, maintainable code.

Streams in Node.js: A Functional Perspective

Streams represent sequences of data chunks flowing asynchronously. Instead of processing the entire dataset at once, streams handle data piece-by-piece, which is efficient and scalable.

Functionally, you can think of streams as **data sources** combined with **pure transformation functions** that process each chunk without side effects. These transformations are composed into a pipeline that handles data flow declaratively.

Composing Stream Transformations as Pure Functions

In Node.js, the `stream.Transform` class lets you create transform streams that process data chunks. When written functionally, these transforms do not mutate state or cause side effects outside the stream, making them reusable and testable.

Example: Creating a Pure Transform Stream

```
const { Transform } = require('stream');

// A transform stream that uppercases text data
function createUppercaseTransform() {
  return new Transform({
    transform(chunk, encoding, callback) {
      // Pure transformation of chunk to uppercase
      const upperChunk = chunk.toString().toUpperCase();
      callback(null, upperChunk);
    }
  });
}
```

Here, `createUppercaseTransform` returns a new transform stream that converts all incoming text to uppercase, without mutating external state.

Reading, Transforming, and Writing Files with Functional Streams

Node.js provides `fs.createReadStream` and `fs.createWriteStream` to read and write files as streams. You can **compose** these with your pure transform streams using the `stream.pipeline` utility, which manages data flow, errors, and backpressure.

Example: Reading a File, Transforming, and Writing Output

```
const fs = require('fs');
const { pipeline } = require('stream');
const { promisify } = require('util');
```

```
const pipelineAsync = promisify(pipeline);

async function processFile(inputPath, outputPath) {
  const uppercaseTransform = createUppercaseTransform();

  try {
    await pipelineAsync(
      fs.createReadStream(inputPath),  // Read stream (source)
      uppercaseTransform,               // Transform stream (pure function)
      fs.createWriteStream(outputPath) // Write stream (sink)
    );
    console.log('File processed successfully.');
  } catch (err) {
    console.error('Pipeline failed:', err);
  }
}

// Usage
processFile('input.txt', 'output.txt');
```

This example demonstrates:

- Composing streams declaratively with `pipeline`.
- Separating pure transformations (`uppercaseTransform`) from side-effecting I/O streams.
- Handling errors cleanly via `pipeline`.

Error Handling and Backpressure

- **Error Handling:** The `pipeline` utility automatically propagates errors from any stream in the chain, simplifying robust error management without try/catch inside every stream.
- **Backpressure:** When the writable destination slows down, readable streams pause to avoid overwhelming buffers. This flow control is built into Node.js streams and works naturally with composed functional streams, preserving efficiency without manual intervention.

Summary

By embracing functional patterns for Node.js stream and file processing, you gain:

- **Modularity:** Pure transform functions can be tested and reused independently.
- **Readability:** Declarative pipelines clarify data flow.
- **Robustness:** `pipeline` handles errors and backpressure elegantly.
- **Performance:** Streaming avoids buffering entire files in memory.

Functional stream processing is a core technique for efficient, maintainable Node.js applications that handle large or continuous data gracefully.

20.3 CLI Tools with Pure Functions

Building command-line interface (CLI) tools in Node.js is a common task. Applying **functional programming principles**—like pure functions, immutability, and functional decomposition—makes CLI tools easier to test, maintain, and extend.

Functional Approach to CLI Tool Development

A CLI tool generally involves these steps:

1. **Parsing command-line arguments**
2. **Processing input data**
3. **Producing output**

In a functional design:

- Each step is handled by pure functions that do not cause side effects (e.g., no direct `console.log` inside processing).
- Side effects like reading input or writing output are confined to a minimal part of the code.
- Data transformations are isolated and reusable.

Parsing Arguments and Processing Data Purely

You can parse CLI arguments and transform input data using pure functions. For example, parsing args into a config object, then using pure functions to transform text or data before outputting results.

Runnable Example: A Functional Text Formatter CLI

This example demonstrates a simple CLI that reads text, converts it to uppercase or lowercase based on an argument, and prints the result.

```
#!/usr/bin/env node

const { argv, stdin, stdout, exit } = require('process');

/**
 * Pure function to parse CLI arguments into a config object.
 * Supports --upper or --lower flags.
 */
function parseArgs(args) {
  return {
    toUpper: args.includes('--upper'),
    toLower: args.includes('--lower')
  };
}

/**
 * Pure function that formats text based on config.
 */
function formatText(text, { toUpper, toLower }) {
  if (toUpper) return text.toUpperCase();
```

```
  if (toLower) return text.toLowerCase();
  return text; // no formatting
}

/**
 * Reads all data from stdin as a Promise.
 */
function readStdin() {
  return new Promise((resolve, reject) => {
    let data = '';
    stdin.setEncoding('utf8');
    stdin.on('data', chunk => data += chunk);
    stdin.on('end', () => resolve(data));
    stdin.on('error', err => reject(err));
  });
}

async function main() {
  const config = parseArgs(argv.slice(2)); // Skip node and script paths
  try {
    const inputText = await readStdin();
    const outputText = formatText(inputText, config);
    stdout.write(outputText);
  } catch (error) {
    console.error('Error:', error);
    exit(1);
  }
}

main();
```

How This Example Embraces Functional Principles

- **Pure functions:**
  - `parseArgs` and `formatText` have no side effects and always return the same output for the same inputs.
  - This makes them easy to test independently.
- **Isolated side effects:**
  - Reading from `stdin` and writing to `stdout` are clearly separated from data processing.
- **Composable and reusable:**
  - The `formatText` function could be reused in other tools or combined with other text transformations.

Benefits of Functional CLI Tools

- **Testability:** Pure functions can be unit tested without mocking I/O.
- **Modularity:** Functions have clear responsibilities and interfaces.
- **Maintainability:** Clear separation between side effects and logic reduces bugs.
- **Reusability:** Logic can be easily reused in different contexts or tools.

Summary

Functional programming patterns help you build CLI tools that are clean, predictable, and easy to maintain. By isolating side effects and focusing on pure functions for argument parsing and data transformation, you create tools that are not only robust but also simple to extend and test. This approach scales well from small utilities to complex command-line applications.

Chapter 21.

Testing Functional Code

1. Testing Pure Functions with Jest
2. Property-Based Testing Concepts
3. Snapshot Testing of Compositions

21 Testing Functional Code

21.1 Testing Pure Functions with Jest

Testing is a crucial part of any software development process, and functional programming makes this easier by encouraging **pure functions**—functions that always produce the same output for the same input and have no side effects. This predictability means tests are straightforward, reliable, and easy to write.

Why Pure Functions Are Ideal for Testing

- **Deterministic**: Given the same inputs, pure functions always return the same output.
- **No Side Effects**: No external state or I/O means fewer things to mock or stub.
- **Isolated Logic**: You can test functions in isolation without complex setup.
- **Simpler Debugging**: Failures point directly to function input/output issues.

Introducing Jest

Jest is a popular JavaScript testing framework designed with simplicity and speed in mind. It supports:

- Easy test writing with descriptive syntax
- Built-in assertions
- Automatic mocks (which can be ignored for pure functions)
- Snapshot testing
- Fast test execution with intelligent parallelization

Writing Simple Jest Tests for Pure Functions

Here's a quick example: suppose we have a pure function that doubles a number.

```
// double.js
function double(x) {
  return x * 2;
}

module.exports = double;
```

Now, write a Jest test to verify it works correctly.

```
// double.test.js
const double = require('./double');

test('doubles positive numbers', () => {
  expect(double(2)).toBe(4);
  expect(double(5)).toBe(10);
});

test('doubles negative numbers', () => {
  expect(double(-3)).toBe(-6);
});
```

```
test('doubles zero', () => {
  expect(double(0)).toBe(0);
});
```

Key Points in This Test Example

- **Test structure**: Use `test()` (or `it()`) to describe the behavior.
- **Assertions**: Use `expect()` with matcher methods like `.toBe()` to check output.
- **Input/Output focus**: Tests only check the return value, no mocks needed.

More Complex Example: Pure Function with Objects

Consider a function that returns a new user object with an updated age:

```
// updateAge.js
function updateAge(user, years) {
  return { ...user, age: user.age + years };
}

module.exports = updateAge;
```

Test:

```
// updateAge.test.js
const updateAge = require('./updateAge');

test('increments user age without mutating original', () => {
  const user = { name: 'Alice', age: 30 };
  const updated = updateAge(user, 5);

  expect(updated.age).toBe(35);
  expect(user.age).toBe(30); // original unchanged
  expect(updated).not.toBe(user); // new object returned
});
```

How Pure Functions Simplify Mocking and Isolation

Since pure functions do not rely on external state or side effects, there's usually no need to mock dependencies or external APIs. This leads to simpler, more stable tests. When your entire logic is composed of pure functions, your tests become focused only on input and output, reducing flakiness and complexity.

Running Tests Efficiently

- Use `jest` CLI command to run tests:
  ```
  npx jest
  ```
- Use `--watch` mode to rerun tests on file changes:
  ```
  npx jest --watch
  ```
- Group related tests in `describe()` blocks for clarity.

- Use clear, descriptive test names for maintainability.

21.1.1 Summary

Testing pure functions with Jest is straightforward and effective due to their deterministic nature and lack of side effects. Writing isolated, focused tests improves code reliability and developer confidence. By combining functional programming principles with Jest's powerful yet simple testing features, you can build robust JavaScript applications with ease.

21.2 Property-Based Testing Concepts

While traditional unit testing checks a function against specific inputs and expected outputs, **property-based testing** takes a broader approach. Instead of testing a handful of cases, it verifies that a function satisfies certain properties or invariants for *many* automatically generated inputs. This helps uncover edge cases and bugs that fixed examples might miss.

Core Concepts of Property-Based Testing

- **Properties**: General truths or rules your function should always satisfy, regardless of input. For example, sorting a list should always produce a list that is ordered and contains the same elements.
- **Generators**: Tools that create random test inputs of various types and sizes. They automatically generate diverse data sets to challenge your code.
- **Shrinking**: When a test fails, the framework tries to minimize the input to the smallest failing example. This helps in understanding and debugging errors efficiently.

Why Use Property-Based Testing?

- **Broader test coverage**: Automatically tests a vast space of inputs, increasing confidence.
- **Fewer false assumptions**: Prevents missing tricky edge cases.
- **Complements unit tests**: Unit tests ensure expected behavior for known cases, while property tests validate behavior over all valid inputs.
- **Great for pure functions**: Since pure functions are deterministic, property testing fits naturally.

Example: Using `fast-check` for Property-Based Testing

`fast-check` is a popular JavaScript property-based testing library that integrates well with Jest.

Suppose we have a pure function that reverses an array:

```
// reverse.js
function reverse(arr) {
  return [...arr].reverse();
}

module.exports = reverse;
```

Here's how we can write a property test to verify some properties about the `reverse` function:

```
// reverse.test.js
const fc = require('fast-check');
const reverse = require('./reverse');

test('reversing twice returns original array', () => {
  fc.assert(
    fc.property(fc.array(fc.anything()), (arr) => {
      // Property: reverse(reverse(arr)) === arr
      return JSON.stringify(reverse(reverse(arr))) === JSON.stringify(arr);
    })
  );
});

test('reversing preserves array length', () => {
  fc.assert(
    fc.property(fc.array(fc.anything()), (arr) => {
      return reverse(arr).length === arr.length;
    })
  );
});
```

Explanation of the Example

- `fc.property` defines a property to test, specifying the input generator (`fc.array(fc.anything())` creates arbitrary arrays).
- `fc.assert` runs many random inputs against the property.
- The first test checks the **involution property**: reversing twice yields the original array.
- The second test asserts the output length remains the same.
- If a failing input is found, `fast-check` automatically shrinks it to the simplest failing case.

Summary

Property-based testing lets you verify that your pure functions uphold important properties across a wide range of inputs, improving the robustness of your code. It complements traditional example-based tests and is especially powerful when combined with tools like Jest and `fast-check` for expressive, automated test suites. Incorporating property-based tests into your workflow can reveal hidden bugs and ensure your functions behave correctly in all scenarios.

21.3 Snapshot Testing of Compositions

Snapshot testing is a powerful technique that captures the **output of a function or component at a specific point in time** and saves it as a snapshot file. Later, when the test runs again, Jest compares the current output against the saved snapshot. If the output changes unexpectedly, the test fails, alerting you to potential regressions.

This approach is especially useful for:

- **Testing complex function compositions** where the output may be too large or intricate for manual assertions.
- **UI components or serialized data**, where you want to ensure the rendered structure stays consistent over time.

Why Use Snapshot Testing?

- **Simplifies tests for large outputs**: You don't have to write verbose assertions for every field or nested structure.
- **Captures integrated behavior**: Tests the result of multiple functions composed together, verifying the entire data flow.
- **Detects unintended changes early**: If an output changes, you get immediate feedback.
- **Easy to update**: If a change is intentional, you can update the snapshot with a simple command.

Using Jest Snapshots to Test Compositions

Let's say you have multiple pure functions that process user data in a pipeline:

```
// userUtils.js
const trimName = (user) => ({
  ...user,
  name: user.name.trim(),
});

const toUpperCaseName = (user) => ({
  ...user,
  name: user.name.toUpperCase(),
});

const addGreeting = (user) => ({
  ...user,
  greeting: `Hello, ${user.name}!`,
});

const processUser = (user) => addGreeting(toUpperCaseName(trimName(user)));

module.exports = { trimName, toUpperCaseName, addGreeting, processUser };
```

Instead of writing multiple assertions for every property, you can write a snapshot test for the composed function's output:

```
// userUtils.test.js
const { processUser } = require('./userUtils');

test('processUser produces expected output', () => {
  const input = { name: '  Alice ' };
  const result = processUser(input);
  expect(result).toMatchSnapshot();
});
```

When you run the test for the first time, Jest creates a snapshot file with the serialized output:

```
// __snapshots__/userUtils.test.js.snap
exports[`processUser produces expected output 1`] = `
Object {
  "greeting": "Hello, ALICE!",
  "name": "ALICE",
}
`;
```

On subsequent test runs, Jest compares the current output to this snapshot and fails if there are any differences.

When Is Snapshot Testing Most Useful?

- When your output is **complex or deeply nested** (e.g., UI render trees, JSON data).
- When testing **compositions of multiple functions** where a single expected output is clearer than multiple assertions.
- When output formatting or structure should be preserved exactly.

Maintaining Snapshot Accuracy

- **Review snapshot changes carefully**: Only update snapshots (`jest --updateSnapshot`) if changes are intentional.
- **Keep snapshots small and focused**: Snapshot entire large objects only if needed, otherwise break down tests.
- **Combine with other test types**: Use snapshot tests alongside unit and property tests for thorough coverage.

21.3.1 Summary

Snapshot testing with Jest offers an elegant way to verify the outputs of complex function compositions or UI renderings without writing verbose assertions. It helps catch regressions quickly and keeps tests maintainable by capturing outputs declaratively. When used thoughtfully, snapshot testing becomes a valuable tool in your functional programming test strategy.

Chapter 22.

Debugging Functional JavaScript

1. Tracing Composed Functions
2. Logging Pipelines and Taps
3. Debugging Recursive Logic

22 Debugging Functional JavaScript

22.1 Tracing Composed Functions

Composed functions — where multiple smaller functions are combined into a single pipeline — are a hallmark of functional programming. They improve readability and modularity but can also make debugging more challenging. Since data flows through many transformations, pinpointing where an issue occurs or verifying intermediate values requires deliberate tracing.

Why Is Tracing Composed Functions Challenging?

- **No intermediate variables**: In functional composition, you often write expressions like `f(g(h(x)))`, so intermediate results aren't assigned to named variables you can easily inspect.
- **Pure functions and immutability**: Because functions don't produce side effects, you can't rely on external logging inside the functions unless explicitly inserted.
- **Error localization**: When a composed pipeline fails or produces unexpected output, it's hard to know which function in the chain is responsible without breaking down the flow.

Strategies for Tracing Without Losing Purity

To debug effectively while maintaining the purity of your functions, you need techniques that allow you to **peek into intermediate data** without altering function signatures or side-effect free behavior.

Using a `tap` Utility Function A common functional helper is `tap`, which takes a value, runs a side-effectful function (like logging), and returns the original value unchanged. This lets you insert debugging steps inside chains without modifying the data flow.

```
const tap = (fn) => (value) => {
  fn(value);
  return value;
};

// Example pipeline
const process = (x) =>
  [x]
    .map(tap((v) => console.log('Initial:', v)))
    .map(x => x + 1)
    .map(tap((v) => console.log('After increment:', v)))
    .map(x => x * 2)
    .map(tap((v) => console.log('After multiply:', v)))
    .pop();

process(5);
```

Output:

```
Initial: 5
After increment: 6
After multiply: 12
```

`tap` helps maintain purity while making data visible at key points.

Breaking Down the Composition Into Named Steps Instead of deeply nesting, break the pipeline into intermediate variables for inspection during debugging:

```
const step1 = (x) => x + 1;
const step2 = (x) => x * 2;

const composed = (x) => step2(step1(x));

const input = 5;
const afterStep1 = step1(input);
console.log('After step1:', afterStep1);
const result = step2(afterStep1);
console.log('Final result:', result);
```

While less elegant, this makes debugging easier, especially during development.

Using Debugging Wrappers You can create a reusable debugging wrapper that logs input and output of any function:

```
const debug = (fn, name = 'fn') => (arg) => {
  console.log(`Entering ${name} with`, arg);
  const result = fn(arg);
  console.log(`Exiting ${name} with`, result);
  return result;
};

const increment = (x) => x + 1;
const double = (x) => x * 2;

const composed = (x) => double(increment(x));

const debuggedComposed = (x) =>
  debug(double, 'double')(debug(increment, 'increment')(x));

debuggedComposed(5);
```

This approach is great for quickly adding logs without changing function internals.

Tools and Techniques to Aid Tracing

- **Browser DevTools and Node Debugger**: Set breakpoints inside individual functions to inspect variables and call stacks.
- **Logging libraries**: Use advanced loggers like `debug` for conditional and leveled logging.
- **Functional debugging utilities**: Libraries like Ramda provide `R.tap` which works similarly to the `tap` example shown.

- **Stepwise evaluation**: Temporarily rewrite pipelines as sequential statements to isolate bugs.
- **Unit tests with detailed assertions**: Isolate functions and test intermediate outputs independently.

22.1.1 Summary

Tracing deeply composed functions requires balancing purity with observability. By inserting side-effectful helpers like `tap`, breaking down pipelines during debugging, and using logging wrappers, you can inspect intermediate results without compromising your functional style. Combined with modern debugging tools, these strategies make functional code more maintainable and easier to debug.

22.2 Logging Pipelines and Taps

When working with functional pipelines, maintaining purity is crucial — meaning functions shouldn't produce side effects like logging directly inside them. But during development or debugging, you often need to peek at intermediate values flowing through your function chains without breaking the flow or mutating data.

This is where the **`tap` combinator** becomes invaluable.

What Is the `tap` Combinator?

`tap` is a small helper function that takes a side-effect function (often a logger) and returns a function which:

- Runs the side-effect with the input value,
- Returns the input value unchanged.

This means you can inject logging (or any side-effect) into your pure pipelines **without breaking immutability or the data flow**.

Writing a Basic `tap` Function

Here's a simple reusable implementation of `tap`:

```
const tap = (fn) => (value) => {
  fn(value);
  return value;
};
```

- `fn` is your side-effect function, such as `console.log`.
- `value` is the data flowing through the pipeline.

- The original `value` is returned unchanged.

Using `tap` in Pipelines

Suppose you have a pipeline that transforms a string by trimming, converting to lowercase, and replacing spaces with hyphens:

```
const trim = (str) => str.trim();
const toLowerCase = (str) => str.toLowerCase();
const replaceSpaces = (str) => str.replace(/\s+/g, '-');

const slugify = (str) =>
  [str]
    .map(tap((v) => console.log('Original input:', v)))
    .map(trim)
    .map(tap((v) => console.log('After trim:', v)))
    .map(toLowerCase)
    .map(tap((v) => console.log('After toLowerCase:', v)))
    .map(replaceSpaces)
    .pop();

slugify('  Hello World From Functional JS  ');
```

Output:

```
Original input:   Hello World From Functional JS
After trim: Hello World From Functional JS
After toLowerCase: hello world from functional js
```

This logging happens *inline* with the pipeline, but the data remains unmodified and immutable.

Customizing `tap` for Contextual Logging

You can build specialized `tap` functions to add context or format logs:

```
const tapLabel = (label) => tap((value) => console.log(`${label}:`, value));

// Usage
const slugifyWithLabels = (str) =>
  [str]
    .map(tapLabel('Input'))
    .map(trim)
    .map(tapLabel('Trimmed'))
    .map(toLowerCase)
    .map(tapLabel('Lowercased'))
    .map(replaceSpaces)
    .pop();

slugifyWithLabels('  Functional JavaScript Rocks!  ');
```

This approach helps keep logs organized and easier to scan.

Integrating `tap` into Composed Functions

If you use `compose` or `pipe` utilities for function composition, you can insert `tap` seamlessly:

```
const compose =
  (...fns) =>
  (x) =>
    fns.reduceRight((v, f) => f(v), x);

const slugifyCompose = compose(
  replaceSpaces,
  tapLabel('After toLowerCase'),
  toLowerCase,
  tapLabel('After trim'),
  trim,
  tapLabel('Original input')
);

slugifyCompose('  Functional JS FTW!  ');
```

Best Practices for Using `tap`

- **Keep logging concise:** Avoid logging entire large objects unnecessarily.
- **Use labels:** Add context to logs for clarity.
- **Remove or disable `tap` in production:** Use environment flags or conditional wrappers.
- **Don't use `tap` to alter data:** It should *only* observe, not transform.
- **Use sparingly:** Insert `tap` calls at strategic points to avoid cluttering logs.

22.2.1 Summary

The `tap` combinator offers a clean, functional way to inject logging or side effects into your otherwise pure function pipelines. It preserves immutability and data flow while helping you observe intermediate values, making debugging and development more straightforward.

By customizing and strategically placing `tap` in your code, you maintain readable, maintainable pipelines without sacrificing the benefits of functional purity.

22.3 Debugging Recursive Logic

Recursive functions are elegant and powerful tools in functional programming, but they come with their own set of debugging challenges. Unlike straightforward loops, recursion involves multiple nested function calls that can quickly become difficult to trace, especially when dealing with incorrect base cases or unexpected input.

Common Pitfalls in Recursive Debugging

- **Infinite recursion**: Happens when base cases are missing or incorrect, causing the function to call itself endlessly until the call stack overflows.
- **Stack overflow errors**: JavaScript environments have a limited call stack size; deep or infinite recursion will cause a `RangeError: Maximum call stack size exceeded`.
- **Unexpected argument values**: Recursive calls with wrong or mutated arguments can derail the function logic.
- **Complex call chains**: Nested recursive calls can be hard to follow without proper traceability.

Techniques for Debugging Recursive Functions

Add Trace Logs with Depth Tracking A common approach is to log the function's arguments and the current recursion depth to visualize the call stack as it grows and shrinks.

Here's an example with a factorial function:

```
function factorial(n, depth = 0) {
  console.log(`${'  '.repeat(depth)}factorial called with n=${n}`);

  if (n <= 1) {
    console.log(`${'  '.repeat(depth)}Reached base case, returning 1`);
    return 1;
  }

  const result = n * factorial(n - 1, depth + 1);
  console.log(`${'  '.repeat(depth)}Returning result=${result} for n=${n}`);
  return result;
}

factorial(4);
```

Output:

```
factorial called with n=4
  factorial called with n=3
    factorial called with n=2
      factorial called with n=1
      Reached base case, returning 1
    Returning result=2 for n=2
  Returning result=6 for n=3
Returning result=24 for n=4
```

Indentation based on `depth` helps visualize the recursive call tree clearly.

Use Debugger Breakpoints Strategically Set breakpoints inside your recursive function with your browser's or Node.js debugger:

- Pause execution on entry to the function.

- Inspect the current arguments and call stack.
- Step into recursive calls and watch variable changes.

This gives a live picture of what's happening inside each recursive step.

Watch for Infinite Recursion If your recursive function never hits the base case, you'll see a stack overflow error:

```
RangeError: Maximum call stack size exceeded
```

To avoid this:

- Double-check base case conditions.
- Confirm that recursive calls approach the base case (e.g., arguments decrement correctly).
- Add guard clauses to prevent invalid inputs.

Test Recursive Functions with Edge Cases Write tests for:

- The smallest input (e.g., `n=0` or empty structures).
- Inputs that should trigger base cases immediately.
- Large inputs to check performance and stack limits.
- Unexpected inputs to confirm error handling.

This testing helps catch bugs early and ensures your recursion handles all scenarios gracefully.

Refactor for Clarity and Maintainability If your recursive function becomes too complex:

- Split logic into smaller helper functions.
- Use descriptive parameter names and comments.
- Consider using iterative or tail-recursive equivalents if supported by your environment.
- Leverage immutable data structures to avoid side effects during recursion.

22.3.1 Example: Recursive Sum of Nested Arrays with Tracing

```
function sumNested(arr, depth = 0) {
  console.log(`${' '.repeat(depth * 2)}sumNested called with`, arr);

  if (!Array.isArray(arr)) {
    console.log(`${' '.repeat(depth * 2)}Base case value:`, arr);
    return arr;
  }

  const result = arr.reduce((acc, val) => acc + sumNested(val, depth + 1), 0);
  console.log(`${' '.repeat(depth * 2)}Returning sum=${result} for`, arr);
```

```
  return result;
}

sumNested([1, [2, [3, 4], 5], 6]);
```

This logs each recursive call's input and output, providing a step-by-step trace.

22.3.2 Summary

Debugging recursive functions in JavaScript requires careful attention to:

- Correct base cases to avoid infinite recursion.
- Using logging and indentation to trace recursive calls.
- Leveraging debugger tools to inspect call stacks and variables.
- Testing edge cases rigorously.
- Refactoring complex recursion for clarity.

With these techniques, you can tame the complexity of recursion and maintain functional, reliable code.

Chapter 23.

Performance Considerations

1. When Functional Code Is Slower or Faster
2. Lazy Evaluation Patterns
3. Optimizing Compositions and Recursion

23 Performance Considerations

23.1 When Functional Code Is Slower or Faster

Functional programming (FP) offers numerous benefits like modularity, readability, and easier reasoning about code. However, it's important to understand how FP techniques can impact performance—sometimes slowing down your programs, and other times enabling optimizations that make them faster.

When Functional Code Can Be Slower

1. **Immutability Overhead**

Functional code encourages immutability—never modifying existing data but creating new copies instead. For large or deeply nested data structures, this copying can add significant overhead:

```
// Imperative mutation (fast, but side effects)
arr.push(4);

// Functional immutable update (creates a new array)
const newArr = [...arr, 4];
```

Repeated creation of new objects or arrays can lead to increased memory usage and slower execution compared to mutating existing structures directly.

2. **Recursive Function Calls**

Recursion is a natural fit for FP but can suffer from stack overhead and potential stack overflow:

```
function factorial(n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
```

Deep recursive calls consume stack frames, slowing down execution versus simple loops. Additionally, many JavaScript engines lack full tail call optimization (TCO), limiting recursion's efficiency.

3. **Excessive Function Creation**

Functional code often composes many small functions, sometimes creating new closures inside loops or pipelines:

```
arr.map(x => x * 2).filter(x => x > 10);
```

Each arrow function allocates a new function object, which can impact performance compared to a single imperative loop with inline logic.

When Functional Code Can Be Faster

1. **Easier Caching and Memoization**

Pure functions with immutable inputs and outputs allow effective caching without side effects:

```
const memoizedFib = memoize(fibonacci);
```

Memoization dramatically speeds up expensive computations, a benefit less straightforward to achieve with imperative code that has hidden side effects.

2. **Compiler and Runtime Optimizations**

Declarative, predictable code lets JavaScript engines optimize more aggressively. For example, pure functions without side effects can be parallelized or reordered safely.

3. **Short-Circuiting and Lazy Evaluation**

Functional libraries and techniques often support lazy evaluation, delaying computation until needed:

```
// Lazy filter and map example (Ramda or Lodash/fp)
const result = Lazy(arr)
  .filter(isValid)
  .map(transform)
  .take(10)
  .value();
```

This reduces unnecessary processing and memory allocations, improving performance especially on large datasets.

Benchmark Example: Sum of Squares

Imperative version:

```
function sumOfSquares(arr) {
  let total = 0;
  for (let i = 0; i < arr.length; i++) {
    total += arr[i] * arr[i];
  }
  return total;
}
```

Functional version:

```
function sumOfSquares(arr) {
  return arr.map(x => x * x).reduce((a, b) => a + b, 0);
}
```

- The imperative version uses a single loop and mutation.
- The functional version creates an intermediate array via `map`, which can increase memory and runtime costs.

For small arrays, performance differences are negligible, but for very large arrays, the imperative approach may be faster unless lazy evaluation is used.

Understanding the Trade-Offs

- **Use functional style for clarity, maintainability, and correctness.**
- **Be mindful of hot code paths where performance is critical.**
- **Optimize with memoization, lazy evaluation, and avoid unnecessary allocations.**
- **Benchmark when in doubt; performance depends on JavaScript engine, data size, and use case.**

23.1.1 Summary

Functional programming may introduce performance overhead through immutability, recursion, and many small function calls, but it also enables optimizations like memoization and lazy evaluation that can boost speed and reduce bugs. By understanding these trade-offs, you can write efficient, maintainable functional code that balances clarity with performance.

23.2 Lazy Evaluation Patterns

Lazy evaluation is a powerful technique in functional programming that can significantly improve performance by **deferring computation until the results are actually needed**. Instead of eagerly computing every value immediately, lazy evaluation calculates values on-demand, avoiding unnecessary work and reducing memory usage.

What Is Lazy Evaluation?

In eager evaluation (the default in JavaScript), expressions are evaluated as soon as they are encountered:

```
const result = [1, 2, 3, 4, 5].map(x => x * 2);
// All elements are processed immediately
```

Lazy evaluation delays this processing until the values are required, which can be particularly beneficial when working with large or infinite data sequences.

Common Lazy Evaluation Techniques in JavaScript

1. **Generators**

Generators produce sequences of values on-the-fly, pausing after each yield and resuming when requested.

```
function* naturalNumbers() {
  let n = 1;
  while (true) {
    yield n++;
  }
}

// Usage:
const gen = naturalNumbers();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

Because values are generated only when requested, generators can represent infinite sequences without exhausting memory.

2. **Iterators**

Iterators are objects that conform to the iterator protocol (`next()` method), often used with generators. They allow controlled, stepwise access to data, enabling lazy consumption.

3. **Lazy Streams**

Lazy streams abstract sequences of data where operations like `map`, `filter`, or `take` are chained but not immediately executed. Only when a terminal operation (e.g., `toArray()`) is called does evaluation happen.

Practical Example: Lazy Filtering and Mapping with Generators

Let's implement a lazy pipeline that filters and maps a sequence of numbers, stopping after a certain condition:

```
function* filter(iterable, predicate) {
  for (const value of iterable) {
    if (predicate(value)) {
      yield value;
    }
  }
}

function* map(iterable, mapper) {
  for (const value of iterable) {
    yield mapper(value);
  }
}

function* take(iterable, limit) {
  let count = 0;
  for (const value of iterable) {
    if (count++ >= limit) break;
    yield value;
  }
}

// Usage:
const numbers = naturalNumbers();
```

```
const processed = take(
  map(
    filter(numbers, n => n % 2 === 0),  // even numbers only
    n => n * n                          // square them
  ),
  5                                     // take first 5 results
);

console.log([...processed]);  // [4, 16, 36, 64, 100]
```

Here, none of the numbers are processed until we iterate with the spread operator. This pipeline avoids processing the entire infinite sequence and only computes the first 5 squared even numbers.

Libraries and Language Features Supporting Lazy Evaluation

- **Lodash/fp** and **Ramda** provide utilities to build composable functional pipelines, but their evaluation is eager by default. However, libraries like **Lazy.js** or **IxJS** extend them with lazy behavior.
- **RxJS** treats streams as asynchronous lazy sequences, allowing you to compose event-driven logic efficiently.
- Native **generators** and **iterators** in JavaScript provide the building blocks for lazy evaluation without external dependencies.

Benefits of Lazy Evaluation

- **Improved performance** by skipping unnecessary computations.
- **Lower memory usage** since values are produced and consumed incrementally.
- **Ability to work with infinite or very large data sets** safely.

23.2.1 Summary

Lazy evaluation defers computation until it's needed, enabling efficient, on-demand data processing. In JavaScript, generators and iterators form the foundation of lazy sequences, and functional libraries build on these concepts to provide powerful, composable pipelines. Leveraging lazy evaluation helps avoid wasted work, reduces memory footprint, and unlocks new ways to handle large or infinite data sets cleanly.

23.3 Optimizing Compositions and Recursion

Functional programming encourages elegant, composable code and recursive solutions. However, without optimization, these patterns can introduce performance overheads. This section

explores practical strategies to optimize composed functions and recursive calls for better runtime efficiency, enabling you to write functional code that's both clean and performant.

Memoization: Caching to Avoid Recomputations

Memoization stores the results of expensive function calls and returns cached results for identical inputs, avoiding redundant computation.

Example: Naive vs Memoized Fibonacci

```
// Naive recursive Fibonacci (exponential time)
function fib(n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

// Memoized Fibonacci (linear time)
function memoize(fn) {
  const cache = new Map();
  return function (arg) {
    if (cache.has(arg)) return cache.get(arg);
    const result = fn(arg);
    cache.set(arg, result);
    return result;
  };
}

const fibMemo = memoize(function fib(n) {
  if (n <= 1) return n;
  return fibMemo(n - 1) + fibMemo(n - 2);
});

console.log(fibMemo(40)); // Efficiently computes the 40th Fibonacci number
```

Memoization is especially effective for recursive functions with overlapping subproblems.

Tail Call Optimization (TCO): Reusing Stack Frames

Tail call optimization allows some recursive calls to execute without adding new stack frames, preventing stack overflow and improving performance.

Tail-recursive factorial example:

```
function factorial(n, acc = 1) {
  if (n <= 1) return acc;
  return factorial(n - 1, n * acc); // Tail call
}
```

While JavaScript engines have inconsistent support for TCO, writing tail-recursive functions prepares your code for future optimizations and can be manually refactored into loops when necessary.

Minimizing Intermediate Data Structures

Chained functional transformations often create many intermediate arrays or objects, which can degrade performance.

Optimization techniques:

- **Fuse operations:** Combine multiple steps into a single pass.
- **Use lazy evaluation:** Use generators or lazy libraries to process items on demand without creating full intermediate collections.

Example: Fusing `map` and `filter`

```
const data = [1, 2, 3, 4, 5];

// Naive: two passes create intermediate arrays
const result = data
  .map(x => x * 2)
  .filter(x => x > 5);

// Fused single pass
const fused = [];
for (const x of data) {
  const mapped = x * 2;
  if (mapped > 5) fused.push(mapped);
}
```

Alternatively, use lazy generators to avoid intermediate arrays (see previous section on lazy evaluation).

Profiling and Identifying Bottlenecks

Before optimizing, profile your code using browser devtools (Chrome DevTools Performance tab, Node.js profiler) to:

- Measure function execution time.
- Identify hot spots caused by deep compositions or recursion.
- Detect excessive memory allocations.

Profiling helps target the most impactful optimizations rather than premature optimization.

Refactoring Example: Improving Recursive Tree Traversal

Naive recursive tree depth calculation:

```
function treeDepth(node) {
  if (!node.children || node.children.length === 0) return 1;
  return 1 + Math.max(...node.children.map(treeDepth));
}
```

Potential issue: Creates intermediate arrays from `map`.

Optimized version without intermediate array:

```
function treeDepthOptimized(node) {
  if (!node.children || node.children.length === 0) return 1;
  let maxDepth = 0;
  for (const child of node.children) {
    const depth = treeDepthOptimized(child);
    if (depth > maxDepth) maxDepth = depth;
  }
  return 1 + maxDepth;
}
```

This reduces memory usage and can improve performance on large trees.

23.3.1 Summary

Optimizing functional compositions and recursion involves:

- **Memoizing** pure functions to cache repeated calls.
- Writing **tail-recursive functions** where possible.
- **Minimizing intermediate data structures** by fusing operations or using lazy evaluation.
- **Profiling your code** to find actual bottlenecks before optimizing.
- Refactoring recursive functions to avoid unnecessary overhead.

Applying these techniques balances functional elegance with practical performance, helping you build robust and efficient JavaScript applications.