Higher-Order Functions in JavaScript

A Comprehensive Guide for Web Developers

Anass Kabil

Version 1.0

January 13, 2026

https://github.com/anassk01/higher-order-functions-js-book

Contents

Part I

Foundations

Chapter 1

What Are Higher-Order Functions?

> *"Functions are first-class citizens in JavaScript—they can go anywhere any other value can go."*
> — Douglas Crockford

1.1 The Big Idea

In JavaScript, functions are **first-class citizens**. This means functions can be:

- Stored in variables
- Passed as arguments to other functions
- Returned from other functions
- Stored in data structures

Definition: Higher-Order Function

A **Higher-Order Function (HOF)** is a function that does at least one of these:

1. Takes one or more functions as arguments
2. Returns a function as its result

That's it. No magic, no complex theory. Just functions working with functions.

1.2 Functions as Values

Before we dive into HOFs, let's cement the foundation: functions are values, just like numbers or strings.

```
// A number stored in a variable
const age = 25;

// A string stored in a variable
const name = ”Alice”;

// A function stored in a variable
const greet = function(person) {
    return `Hello, ${person}!`;
};

// All three are just values
console.log(age);         // 25
console.log(name);        // Alice
console.log(greet);       // [Function: greet]
console.log(greet(name)); // Hello, Alice!
```

Key Insight

The function `greet` is not special—it's a value that happens to be callable.

1.2.1 Arrow Functions: The Modern Syntax

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

// Arrow function (equivalent)
const addArrow = (a, b) ⇒ a + b;

// Single parameter: parentheses optional
const double = x ⇒ x * 2;

// No parameters: empty parentheses required
const sayHi = () ⇒ ”Hi!”;
```

Arrow functions are not just shorter syntax—they also don't have their own `this` binding, which matters in web development (we'll cover this in Chapter 9).

1.3 Functions That Take Functions

The most common HOF pattern: passing a function as an argument.

1.3.1 The Callback Pattern

```
// A higher-order function: takes a function as argument
function processUser(user, callback) {
    // Do some processing
    const processed = {
        ...user,
        processedAt: Date.now()
    };

    // Call the passed-in function with the result
    callback(processed);
}

// Usage: we pass a function as the second argument
processUser({ name: ”Bob” }, function(result) {
    console.log(result);
});
```

```
// With arrow function (cleaner)
processUser({ name: ”Bob” }, result => console.log(result));
```

Why is `processUser` a higher-order function? Because it takes `callback` as a parameter, and `callback` is a function.

1.3.2 Real Web Dev Example: Event Handlers

Every time you add an event listener, you're using a higher-order function:

```
// addEventListener is a HOF--it takes a function as its second argument
button.addEventListener('click', function(event) {
    console.log('Button clicked!');
});

// With arrow function
button.addEventListener('click', (event) => {
    console.log('Button clicked!');
});

// Or pass a named function
function handleClick(event) {
    console.log('Button clicked!');
}
button.addEventListener('click', handleClick);
```

`addEventListener` doesn't know or care what your function does—it just knows to call it when the event occurs.

1.4 Functions That Return Functions

The second HOF pattern: a function that creates and returns another function.

1.4.1 The Factory Pattern

```
// A function that returns a function
function createMultiplier(factor) {
    // This inner function is returned
    return function(number) {
        return number * factor;
    };
}

// Create specific multiplier functions
const double = createMultiplier(2);
const triple = createMultiplier(3);
const tenX = createMultiplier(10);

// Use them
console.log(double(5));  // 10
console.log(triple(5));  // 15
console.log(tenX(5));    // 50
```

1. `createMultiplier(2)` runs and returns a new function
2. That returned function "remembers" that `factor` is `2`
3. We store that function in `double`
4. When we call `double(5)`, it multiplies `5 * 2`

Key Insight

This "remembering" is called a **closure**—the inner function closes over the variable `factor` from its outer scope.

1.4.2 With Arrow Functions

```
// Same thing, more concise
const createMultiplier = factor => number => number * factor;

// Reading this:
// createMultiplier is a function that takes `factor`
// and returns a function that takes `number`
// and returns `number * factor`

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

1.4.3 Real Web Dev Example: Configured Functions

API Fetcher Factory

```
// Create a fetcher for a specific API
function createApiFetcher(baseUrl) {
    return function(endpoint) {
        return fetch(`${baseUrl}${endpoint}`)
            .then(response => response.json());
    };
}

// Create fetchers for different APIs
const githubApi = createApiFetcher('https://api.github.com');
const myApi = createApiFetcher('https://api.myapp.com');

// Use them
```

```
githubApi('/users/octocat').then(data => console.log(data));
myApi('/products').then(data => console.log(data));
```

1.5 Both Patterns Combined

Many HOFs both take AND return functions:

```
// Takes a function, returns a function
function withLogging(fn) {
    return function(...args) {
        console.log(`Calling with args:`, args);
        const result = fn(...args);
        console.log(`Result:`, result);
        return result;
    };
}

// Original function
const add = (a, b) => a + b;

// Enhanced function
const addWithLogging = withLogging(add);

addWithLogging(2, 3);
// Logs: Calling with args: [2, 3]
// Logs: Result: 5
// Returns: 5
```

Key Insight

This pattern—wrapping a function to add behavior—is foundational to middleware, decorators, and much of web development.

1.6 Mental Model: Functions as Values

Here’s the mental shift that makes HOFs intuitive:

Think of functions like you think of objects.

With Objects	With Functions
`const user = { name: ”Bob” }`	`const greet = () => ”Hi”`
`doSomething(user)`	`doSomething(greet)`
`return user`	`return greet`
`users.push(user)`	`handlers.push(greet)`

When someone says “pass a function,” think “pass a value that happens to be callable.”

1.7 Why Higher-Order Functions Matter in Web Dev

1.7.1 Problem: Repetitive Patterns

× Bad

```
// Without HOFs: repetitive code
const numbers = [1, 2, 3, 4, 5];

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

// Triple each number
const tripled = [];
for (let i = 0; i < numbers.length; i++) {
    tripled.push(numbers[i] * 3);
}

// Get string versions
const strings = [];
for (let i = 0; i < numbers.length; i++) {
    strings.push(String(numbers[i]));
}
```

1.7.2 Solution: Abstract the Pattern

Good

```
// With HOFs: the pattern is abstracted
const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map(n => n * 2);
const tripled = numbers.map(n => n * 3);
const strings = numbers.map(n => String(n));
```

Key Insight

`map` is a HOF that abstracts "do something to each element." We just tell it *what* to do.

1.8 The Three Questions

When you see a function, ask:

1. **Does it take a function as an argument?** → It's a HOF
2. **Does it return a function?** → It's a HOF
3. **Does it do both?** → It's a HOF

```javascript
// Takes function: YES (the callback)
// Returns function: NO
array.forEach(callback);

// Takes function: NO
// Returns function: YES
function createHandler() {
    return () => console.log('handled');
}

// Takes function: YES (fn)
// Returns function: YES (the wrapper)
function memoize(fn) {
    const cache = {};
    return (arg) => {
        if (!(arg in cache)) {
            cache[arg] = fn(arg);
        }
        return cache[arg];
    };
}
```

1.9 Practice Exercises

Exercise 1.1: Identify the HOFs

Which of these are higher-order functions? Why?

Exercise

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

// B
function runTwice(fn) {
    fn();
    fn();
}

// C
const numbers = [1, 2, 3];
numbers.push(4);

// D
function createCounter() {
    let count = 0;
    return function() {
        return ++count;
    };
}

// E
setTimeout(function() {
    console.log('delayed');
}, 1000);
```

Answers:

- **A:** Not a HOF (doesn't take or return functions)
- **B:** HOF (takes a function as argument)
- **C:** Not a HOF (`push` doesn't involve functions here)
- **D:** HOF (returns a function)
- **E:** `setTimeout` is a HOF (takes a function as argument)

Exercise 1.2: Convert to HOF

Convert this repetitive code using a higher-order function:

Exercise

```javascript
// Current code: three similar functions
function logError(message) {
    console.log(`[ERROR] ${message}`);
}

function logWarning(message) {
    console.log(`[WARNING] ${message}`);
}

function logInfo(message) {
    console.log(`[INFO] ${message}`);
}
```

Solution:

```javascript
function createLogger(level) {
    return function(message) {
        console.log(`[${level}] ${message}`);
    };
}

const logError = createLogger('ERROR');
const logWarning = createLogger('WARNING');
const logInfo = createLogger('INFO');
```

Exercise 1.3: Event Handler Factory

Create a HOF called `createClickHandler` that:

- Takes an `action` string
- Returns a function suitable for use as a click handler
- The returned function should log: `”Button clicked: {action}”`

Exercise

```javascript
// Your solution:
const createClickHandler = action => () => {
    console.log(`Button clicked: ${action}`);
};

// Usage:
const saveHandler = createClickHandler('save');
const deleteHandler = createClickHandler('delete');

button1.addEventListener('click', saveHandler);
button2.addEventListener('click', deleteHandler);
```

1.10 Chapter Summary

Chapter Summary

Pattern	Description
First-class function	A function treated as a value
Higher-order function	A function that takes/returns functions
Callback	A function passed to another function
Factory function	A function that returns a function
Closure	A function that remembers its outer scope

Key Insight

HOFs let us abstract *what varies* (the operation) from *what stays the same* (the pattern/structure).

Chapter 2

The Core Three — map, filter, reduce

> *"Give me a lever long enough and a fulcrum on which to place it, and I shall move the world."*
>
> — Archimedes

These three array methods are the workhorses of functional JavaScript. Master them, and you'll write cleaner, more expressive code.

2.1 map: Transform Every Element

`map` creates a new array by applying a function to every element.

Definition: map

```
const newArray = originalArray.map(transformFunction);
```

2.1.1 Basic Examples

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

// Double each number
const doubled = numbers.map(n => n * 2);
// [2, 4, 6, 8, 10]

// Square each number
const squared = numbers.map(n => n * n);
// [1, 4, 9, 16, 25]

// Convert to strings
const strings = numbers.map(n => String(n));
// ['1', '2', '3', '4', '5']
```

2.1.2 The Transformation Function

The function you pass to `map` receives three arguments:

```
array.map((element, index, originalArray) => {
    // element: the current item
    // index: its position (0, 1, 2...)
    // originalArray: the whole array (rarely used)
    return transformedValue;
});
```

Usually, you only need `element`:

```
const prices = [10, 20, 30];
const withTax = prices.map(price => price * 1.2);
// [12, 24, 36]
```

Sometimes you need the index:

```
const letters = ['a', 'b', 'c'];
const numbered = letters.map((letter, index) => `${index + 1}. ${letter}`);
// ['1. a', '2. b', '3. c']
```

2.1.3 Web Dev Example: Processing API Data

API Response Transformation

```
// API returns array of user objects
const apiResponse = [
    { id: 1, first_name: 'John', last_name: 'Doe', email: 'john@example.com' },
    { id: 2, first_name: 'Jane', last_name: 'Smith', email: 'jane@example.com' }
];

// Transform to the shape your UI needs
const users = apiResponse.map(user => ({
    id: user.id,
    fullName: `${user.first_name} ${user.last_name}`,
    email: user.email.toLowerCase()
}));

// [
//   { id: 1, fullName: 'John Doe', email: 'john@example.com' },
//   { id: 2, fullName: 'Jane Smith', email: 'jane@example.com' }
// ]
```

2.1.4 Web Dev Example: Rendering Lists

```
// React pattern
const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' }
];

const userElements = users.map(user => (
    <li key={user.id}>{user.name}</li>
));

// Vanilla JS pattern
```

```
const userHTML = users.map(user => `<li>${user.name}</li>`).join('');
document.querySelector('ul').innerHTML = userHTML;
```

Critical Rule: `map` always returns an array with the same number of elements:

- Input: 5 elements → Output: 5 elements
- Input: 0 elements → Output: 0 elements

If you need fewer elements, that's `filter`. If you need more or fewer, that might be `flatMap` or `reduce`.

2.2 filter: Select Elements That Pass a Test

`filter` creates a new array containing only elements that pass a test.

Definition: filter

```
const newArray = originalArray.filter(testFunction);
```

The test function (predicate) must return `true` or `false`:

- Return `true` → element is included
- Return `false` → element is excluded

2.2.1 Basic Examples

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Keep only even numbers
const evens = numbers.filter(n => n % 2 === 0);
// [2, 4, 6, 8, 10]

// Keep only numbers greater than 5
const big = numbers.filter(n => n > 5);
// [6, 7, 8, 9, 10]

// Keep only numbers between 3 and 7
const middle = numbers.filter(n => n >= 3 && n <= 7);
// [3, 4, 5, 6, 7]
```

2.2.2 The Predicate Function

A **predicate** is a function that returns `true` or `false`:

```
// These are predicates
const isEven = n => n % 2 === 0;
const isPositive = n => n > 0;
const isLongEnough = str => str.length >= 3;
const hasEmail = user => user.email !== undefined;
```

Using named predicates makes code self-documenting:

× **Bad**

```
const numbers = [-2, -1, 0, 1, 2, 3, 4];

// Less clear
const result = numbers.filter(n => n > 0 && n % 2 === 0);
```

Good

```
// More clear
const isPositive = n => n > 0;
const isEven = n => n % 2 === 0;
const isPositiveAndEven = n => isPositive(n) && isEven(n);

const result = numbers.filter(isPositiveAndEven);
// [2, 4]
```

2.2.3 Web Dev Example: Filtering Products

```
const products = [
    { id: 1, name: 'Laptop', price: 999, inStock: true, category: 'electronics' },
    { id: 2, name: 'Shirt', price: 29, inStock: true, category: 'clothing' },
    { id: 3, name: 'Phone', price: 699, inStock: false, category: 'electronics' },
    { id: 4, name: 'Pants', price: 59, inStock: true, category: 'clothing' }
];

// Only in-stock items
const available = products.filter(p => p.inStock);

// Only electronics
const electronics = products.filter(p => p.category === 'electronics');

// Affordable and available
const affordable = products.filter(p => p.price < 100 && p.inStock);
// [{ id: 2, ... }, { id: 4, ... }]
```

2.2.4 Web Dev Example: Form Validation

```
const formFields = [
    { name: 'email', value: 'test@example.com', required: true },
    { name: 'phone', value: '', required: false },
    { name: 'name', value: '', required: true }
];

// Find all invalid required fields
const invalidFields = formFields.filter(field =>
    field.required && field.value.trim() === ''
);

// [{ name: 'name', value: '', required: true }]

if (invalidFields.length > 0) {
    console.log('Please fill in:', invalidFields.map(f => f.name));
}
```

2.2.5 Filtering Out Falsy Values

A common pattern to clean up arrays:

```
const messyArray = [0, 'hello', '', null, 'world', undefined, 42, false];

// Remove all falsy values
const clean = messyArray.filter(Boolean);
// ['hello', 'world', 42]

// This works because Boolean(value) returns true/false
// filter keeps elements where Boolean(element) is true
```

The `Boolean` function is a handy predicate for removing falsy values like `null`, `undefined`, `”`, `0`, and `false`.

2.3 reduce: Accumulate to a Single Value

`reduce` processes an array and accumulates it into a single value. That value can be a number, string, object, or even another array.

Definition: reduce

```
const result = array.reduce(reducerFunction, initialValue);
```

The reducer function receives:

```
array.reduce((accumulator, currentElement, index, array) => {
    // accumulator: the running total/result
    // currentElement: the current item being processed
    // Return the new accumulator value
    return newAccumulator;
}, initialValue);
```

2.3.1 Basic Example: Sum

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

const sum = numbers.reduce((total, num) => total + num, 0);
// Step by step:
// total=0, num=1 -> return 1
// total=1, num=2 -> return 3
// total=3, num=3 -> return 6
// total=6, num=4 -> return 10
// total=10, num=5 -> return 15
// Result: 15
```

```
Step-by-step execution of reduce:
Initial: accumulator = 0
Step 1: 0 + 1 = 1 (accumulator is now 1) Step 2: 1 + 2 = 3 (accumulator is now 3)
Step 3: 3 + 3 = 6 (accumulator is now 6) Step 4: 6 + 4 = 10 (accumulator is now 10)
Step 5: 10 + 5 = 15 (accumulator is now 15)
Final: 15
```

Always Provide Initial Value!

```
// GOOD: explicit initial value
const sum = numbers.reduce((acc, n) => acc + n, 0);

// RISKY: no initial value (uses first element)
const sum = numbers.reduce((acc, n) => acc + n);
// Works, but fails on empty array!

// This throws an error:
[].reduce((acc, n) => acc + n); // TypeError!

// This returns 0 safely:
[].reduce((acc, n) => acc + n, 0); // 0
```

;

2.3.2 Common Reductions

b// Sum const sum = numbers.reduce((acc, n) => acc + n, 0); // 15

// Product const product = numbers.reduce((acc, n) => acc * n, 1); // 120

// Maximum const max = numbers.reduce((acc, n) => n > acc ? n : acc, -Infinity); // 5

// Minimum const min = numbers.reduce((acc, n) => n < acc ? n : acc, Infinity); // 1

2.3.3 reduce to Object

Creating a Lookup Object

```
const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Charlie' }
];

// Create a lookup object by ID
const usersById = users.reduce((acc, user) => {
    acc[user.id] = user;
    return acc;
}, {});

// {
//   1: { id: 1, name: 'Alice' },
//   2: { id: 2, name: 'Bob' },
//   3: { id: 3, name: 'Charlie' }
// }

// Now O(1) lookup instead of O(n) find:
const user = usersById[2]; // { id: 2, name: 'Bob' }
```

2.3.4 reduce to Count Occurrences

```
const fruits = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple'];

const counts = fruits.reduce((acc, fruit) => {
    acc[fruit] = (acc[fruit] || 0) + 1;
    return acc;
}, {});

// { apple: 3, banana: 2, orange: 1 }
```

2.3.5 reduce to Group By

```
const people = [
    { name: 'Alice', department: 'Engineering' },
    { name: 'Bob', department: 'Sales' },
    { name: 'Charlie', department: 'Engineering' },
    { name: 'Diana', department: 'Sales' }
];

const byDepartment = people.reduce((acc, person) => {
    const dept = person.department;
    if (!acc[dept]) {
        acc[dept] = [];
    }
    acc[dept].push(person);
    return acc;
}, {});
```

```
// {
//   Engineering: [{ name: 'Alice', ... }, { name: 'Charlie', ... }],
//   Sales: [{ name: 'Bob', ... }, { name: 'Diana', ... }]
// }
```

2.3.6 Web Dev Example: Shopping Cart Total

```
const cart = [
    { name: 'Laptop', price: 999, quantity: 1 },
    { name: 'Mouse', price: 29, quantity: 2 },
    { name: 'Keyboard', price: 79, quantity: 1 }
];

const total = cart.reduce((sum, item) => {
    return sum + (item.price * item.quantity);
}, 0);

// 999 + 58 + 79 = 1136
```

2.3.7 Web Dev Example: Flatten Query Parameters

```
const params = [
    { key: 'page', value: '1' },
    { key: 'sort', value: 'name' },
    { key: 'order', value: 'asc' }
];

const queryString = params.reduce((acc, param, index) => {
    const prefix = index === 0 ? '?' : '&';
    return `${acc}${prefix}${param.key}=${param.value}`;
}, '');

// '?page=1&sort=name&order=asc'
```

2.4 Method Chaining: Combining map, filter, reduce

The real power comes from combining these methods:

```
const users = [
    { name: 'Alice', age: 25, active: true },
    { name: 'Bob', age: 17, active: true },
    { name: 'Charlie', age: 30, active: false },
    { name: 'Diana', age: 22, active: true }
];

// Get names of active adults
const activeAdultNames = users
    .filter(user => user.active)          // Keep active users
    .filter(user => user.age >= 18)       // Keep adults
    .map(user => user.name);              // Extract names
```

```
// ['Alice', 'Diana']
```

2.4.1 Reading Chains: Top to Bottom

```
const result = data
    .filter(...)   // First: remove unwanted items
    .map(...)      // Second: transform remaining items
    .reduce(...);  // Third: combine into final result

// Read it as a story:
// "Take data, keep only X, transform each to Y, combine into Z"
```

Key Insight

Think of data processing as a **pipeline**—data flows through transformations, filters, and accumulations.

2.4.2 Web Dev Example: API Response Processing

```
// Raw API response
const apiResponse = {
    data: [
        { id: 1, type: 'user', attributes: { name: 'Alice', email: 'alice@test.com',
          ↪  role: 'admin' }},
        { id: 2, type: 'user', attributes: { name: 'Bob', email: 'bob@test.com',
          ↪  role: 'user' }},
        { id: 3, type: 'user', attributes: { name: 'Charlie', email: null, role:
          ↪  'user' }},
        { id: 4, type: 'user', attributes: { name: 'Diana', email: 'diana@test.com',
          ↪  role: 'admin' }}
    ]
};

// Process: get admin emails for notification
const adminEmails = apiResponse.data
    .filter(item => item.attributes.role === 'admin')  // Only admins
    .filter(item => item.attributes.email !== null)    // Has email
    .map(item => item.attributes.email);               // Extract email

// ['alice@test.com', 'diana@test.com']
```

2.5 Performance Consideration

Each `map` and `filter` creates a new array:

```javascript
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const result = numbers
    .filter(n => n % 2 === 0)   // Creates [2, 4, 6, 8, 10]
    .map(n => n * 2)            // Creates [4, 8, 12, 16, 20]
    .filter(n => n > 10);       // Creates [12, 16, 20]
```

For small arrays (< 10,000 items): Don't worry. The clarity is worth it.
For large arrays: We'll cover optimization in Chapter 12 (transducers, lazy evaluation).

2.6 When to Use Each

Goal	Method	Example
Transform each element	`map`	Convert prices to include tax
Remove some elements	`filter`	Keep only in-stock items
Combine into one value	`reduce`	Calculate total price
Find one element	`find`	Get user by ID (Chapter 3)
Check if any pass	`some`	Are any items on sale? (Chapter 3)
Check if all pass	`every`	Are all fields valid? (Chapter 3)

2.7 Practice Exercises

Exercise 2.1: Data Transformation

Given this API response, extract an array of formatted strings:

Exercise

```javascript
const products = [
    { id: 1, name: 'Laptop', price: 999.99, currency: 'USD' },
    { id: 2, name: 'Mouse', price: 29.99, currency: 'USD' },
    { id: 3, name: 'Keyboard', price: 79.99, currency: 'USD' }
];

// Solution:
const formatted = products.map(p => `${p.name}: $${p.price}`);
// ['Laptop: $999.99', 'Mouse: $29.99', 'Keyboard: $79.99']
```

Exercise 2.2: Filter Chain

Given this data, find all active premium users from the US:

Exercise

```javascript
const users = [
    { id: 1, name: 'Alice', country: 'US', plan: 'premium', active: true },
    { id: 2, name: 'Bob', country: 'UK', plan: 'premium', active: true },
    { id: 3, name: 'Charlie', country: 'US', plan: 'free', active: true },
    { id: 4, name: 'Diana', country: 'US', plan: 'premium', active: false },
    { id: 5, name: 'Eve', country: 'US', plan: 'premium', active: true }
];

// Solution:
const result = users
    .filter(u => u.active)
    .filter(u => u.plan === 'premium')
    .filter(u => u.country === 'US');
// [{ id: 1, ... }, { id: 5, ... }]
```

Exercise 2.3: Reduce to Object

Convert this array to an object grouped by category:

Exercise

```javascript
const items = [
    { name: 'Apple', category: 'fruit' },
    { name: 'Carrot', category: 'vegetable' },
    { name: 'Banana', category: 'fruit' },
    { name: 'Broccoli', category: 'vegetable' }
];

// Solution:
const grouped = items.reduce((acc, item) => {
    if (!acc[item.category]) {
        acc[item.category] = [];
    }
    acc[item.category].push(item.name);
    return acc;
}, {});
// { fruit: ['Apple', 'Banana'], vegetable: ['Carrot', 'Broccoli'] }
```

Exercise 2.4: Complete Pipeline

Build a data processing pipeline for this e-commerce scenario:

Exercise

```
const orders = [
    { id: 1, customer: 'Alice', items: [{ price: 10 }, { price: 20 }], status:
      ↪  'completed' },
    { id: 2, customer: 'Bob', items: [{ price: 15 }], status: 'pending' },
    { id: 3, customer: 'Alice', items: [{ price: 30 }, { price: 40 }], status:
      ↪  'completed' },
    { id: 4, customer: 'Charlie', items: [{ price: 25 }], status: 'completed' }
];

// Task: Calculate total revenue from completed orders
// Solution:
const revenue = orders
    .filter(order => order.status === 'completed')
    .flatMap(order => order.items)
    .reduce((sum, item) => sum + item.price, 0);
// 10 + 20 + 30 + 40 + 25 = 125
```

2.8 Chapter Summary

Chapter Summary

Method	Input	Output	Purpose
`map`	Array of N items	Array of N items	Transform each element
`filter`	Array of N items	Array of 0 to N items	Keep elements that pass test
`reduce`	Array of N items	Single value (any type)	Accumulate to one result

Key Insight

Think of data processing as a pipeline—data flows through transformations, filters, and accumulations.

Chapter 3

Beyond the Core — find, some, every, flatMap

> *"Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away."*
>
> — Antoine de Saint-Exupéry

The core three (`map`, `filter`, `reduce`) handle most cases, but these additional methods make specific patterns cleaner and more efficient.

3.1 find: Get the First Match

`find` returns the **first element** that passes a test, or `undefined` if none match.

Definition: find

```
const element = array.find(testFunction);
// Returns: the element itself, or undefined
```

3.1.1 Basic Example

```
const numbers = [1, 5, 10, 15, 20];

const firstBigNumber = numbers.find(n => n > 8);
// 10 (not [10, 15, 20] -- just the first one)

const notFound = numbers.find(n => n > 100);
// undefined
```

3.1.2 find vs filter

Method	Returns	Use When
`filter`	Array of all matches	You need all matching items
`find`	First match or `undefined`	You need just one item

```
const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Alice' }  // Duplicate name
];

// filter: all users named Alice
users.filter(u => u.name === 'Alice');
// [{ id: 1, name: 'Alice' }, { id: 3, name: 'Alice' }]

// find: first user named Alice
users.find(u => u.name === 'Alice');
// { id: 1, name: 'Alice' }
```

3.1.3 Performance: find Stops Early

`find` stops iterating as soon as it finds a match:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// find stops at 3
numbers.find(n => {
    console.log('Checking', n);
    return n > 2;
});
// Logs: Checking 1, Checking 2, Checking 3
// Returns: 3

// filter checks everything
numbers.filter(n => {
    console.log('Checking', n);
    return n > 2;
});
// Logs: Checking 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
// Returns: [3, 4, 5, 6, 7, 8, 9, 10]
```

> Use `find` when you only need one item—it's more efficient than `filter` because it stops at the first match.

3.1.4 Web Dev Example: Find User by ID

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

function getUserById(id) {
    return users.find(user => user.id === id);
}

const user = getUserById(2);
// { id: 2, name: 'Bob', email: 'bob@test.com' }
```

```
const notFound = getUserById(999);
// undefined
```

3.1.5 Handling undefined

```
const user = users.find(u => u.id === 999);

// WRONG: might crash
console.log(user.name); // TypeError: Cannot read property 'name' of undefined

// RIGHT: check first
if (user) {
    console.log(user.name);
}

// Or use optional chaining
console.log(user?.name); // undefined (no crash)

// Or provide default
const name = user?.name ?? 'Unknown';
```

3.1.6 findIndex: Get Position Instead of Element

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

const index = numbers.findIndex(n => n > 25);
// 2 (position of 30)

const notFoundIndex = numbers.findIndex(n => n > 100);
// -1 (not found)
```

Useful when you need to modify or remove:

```
const todos = [
    { id: 1, text: 'Learn JS', done: false },
    { id: 2, text: 'Build app', done: false }
];

// Find and update
const index = todos.findIndex(t => t.id === 1);
if (index !== -1) {
    todos[index] = { ...todos[index], done: true };
}
```

3.2 some: Does Any Element Pass?

`some` returns `true` if **at least one** element passes the test.

Definition: some

```
const hasMatch = array.some(testFunction);
// Returns: true or false
```

3.2.1 Basic Examples

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

numbers.some(n => n > 4);     // true (5 is > 4)
numbers.some(n => n > 10);    // false (none are > 10)
numbers.some(n => n === 3);   // true (3 exists)
```

3.2.2 some Stops Early

Like `find`, `some` stops as soon as it finds a passing element:

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

numbers.some(n => {
    console.log('Checking', n);
    return n === 2;
});
// Logs: Checking 1, Checking 2
// Returns: true (stopped early)
```

3.2.3 Web Dev Example: Permission Check

```
const user = {
    name: 'Alice',
    roles: ['editor', 'viewer']
};

const adminRoles = ['admin', 'superadmin'];

const isAdmin = user.roles.some(role => adminRoles.includes(role));
// false

// Or check for specific permission
const canEdit = user.roles.some(role => role === 'editor' || role === 'admin');
// true
```

3.2.4 Web Dev Example: Form Has Errors

```
const formFields = [
    { name: 'email', error: null },
    { name: 'password', error: 'Too short' },
    { name: 'username', error: null }
];

const hasErrors = formFields.some(field => field.error !== null);
// true

if (hasErrors) {
    console.log('Please fix errors before submitting');
}
```

3.3 every: Do All Elements Pass?

`every` returns `true` only if **all elements** pass the test.

Definition: every

```
const allPass = array.every(testFunction);
// Returns: true or false
```

3.3.1 Basic Examples

```
const numbers = [2, 4, 6, 8];

numbers.every(n => n % 2 === 0);   // true (all even)
numbers.every(n => n > 5);         // false (2 and 4 fail)
numbers.every(n => n > 0);         // true (all positive)
```

3.3.2 every Stops Early (On First Failure)

```
const numbers = [2, 4, 5, 6, 8];  // 5 is odd

numbers.every(n => {
    console.log('Checking', n);
    return n % 2 === 0;
});
// Logs: Checking 2, Checking 4, Checking 5
// Returns: false (stopped at 5)
```

Empty Array: `every` returns `true`

```
.every(n => n > 0);  // true (vacuous truth)
```

This might surprise you. Logically: “all elements pass” is true when there are no elements to fail.

3.3.3 Web Dev Example: Form Validation

```
const formFields = [
    { name: 'email', value: 'test@example.com', valid: true },
    { name: 'password', value: '12345678', valid: true },
    { name: 'username', value: 'alice', valid: true }
];

const isFormValid = formFields.every(field => field.valid);
// true

// With validation functions
const validators = {
    email: value => value.includes('@'),
    password: value => value.length >= 8,
    username: value => value.length >= 3
};

const isValid = formFields.every(field =>
    validators[field.name](field.value)
);
```

3.3.4 Web Dev Example: All Items Selected

```
const items = [
    { id: 1, selected: true },
    { id: 2, selected: true },
    { id: 3, selected: false }
];

const allSelected = items.every(item => item.selected);
// false

// Toggle ”select all” checkbox state
selectAllCheckbox.checked = allSelected;
```

3.4 some vs every: Quick Reference

Method	Returns true when	Stops when	Empty array
`some`	At least one passes	First pass	`false`
`every`	All pass	First fail	`true`

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

// ”Is there ANY number greater than 3?”
numbers.some(n => n > 3);    // true

// ”Are ALL numbers greater than 3?”
numbers.every(n => n > 3);  // false
```

3.5 flatMap: Map and Flatten in One Step

`flatMap` combines `map` and `flat(1)`—it transforms each element and flattens one level.

3.5.1 The Problem flatMap Solves

```
const sentences = ['Hello world', 'How are you'];

// map gives nested arrays
const words = sentences.map(s => s.split(' '));
// [['Hello', 'world'], ['How', 'are', 'you']]

// We need to flatten
const flatWords = sentences.map(s => s.split(' ')).flat();
// ['Hello', 'world', 'How', 'are', 'you']

// flatMap does both in one step
const words2 = sentences.flatMap(s => s.split(' '));
// ['Hello', 'world', 'How', 'are', 'you']
```

Definition: flatMap

```
// These are equivalent:
array.flatMap(fn)
array.map(fn).flat(1)
```

Use `flatMap` when your transformation function returns an array and you want all results in a single flat array.

3.5.2 When Your Transform Returns an Array

```
const users = [
    { name: 'Alice', pets: ['cat', 'dog'] },
    { name: 'Bob', pets: ['fish'] },
    { name: 'Charlie', pets: [] }
];

// Get all pets
const allPets = users.flatMap(user => user.pets);
// ['cat', 'dog', 'fish']

// With map, you'd get nested arrays:
const nested = users.map(user => user.pets);
// [['cat', 'dog'], ['fish'], []]
```

3.5.3 flatMap for Conditional Inclusion

Return empty array `[]` to exclude items, array with item `[item]` to include:

```javascript
const numbers = [1, 2, 3, 4, 5, 6];

// Double only even numbers, remove odds
const doubledEvens = numbers.flatMap(n =>
    n % 2 === 0 ? [n * 2] : []
);
// [4, 8, 12]

// Equivalent to filter + map:
const same = numbers.filter(n => n % 2 === 0).map(n => n * 2);
```

3.5.4 Web Dev Example: Nested Data Extraction

```javascript
const departments = [
    {
        name: 'Engineering',
        teams: [
            { name: 'Frontend', members: ['Alice', 'Bob'] },
            { name: 'Backend', members: ['Charlie'] }
        ]
    },
    {
        name: 'Design',
        teams: [
            { name: 'UX', members: ['Diana', 'Eve'] }
        ]
    }
];

// Get all team names
const teamNames = departments.flatMap(dept =>
    dept.teams.map(team => team.name)
);
// ['Frontend', 'Backend', 'UX']

// Get all members
const allMembers = departments.flatMap(dept =>
    dept.teams.flatMap(team => team.members)
);
// ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve']
```

3.5.5 Web Dev Example: Processing Nested API Response

```javascript
const apiResponse = {
    pages: [
        { items: [{ id: 1 }, { id: 2 }] },
        { items: [{ id: 3 }] },
        { items: [{ id: 4 }, { id: 5 }] }
    ]
};

// Flatten all items from all pages
const allItems = apiResponse.pages.flatMap(page => page.items);
```

```
// [{ id: 1 }, { id: 2 }, { id: 3 }, { id: 4 }, { id: 5 }]
```

3.6 Combining Everything: Real Patterns

3.6.1 Pattern: Find and Transform

```
const products = [
    { id: 1, name: 'Laptop', price: 999 },
    { id: 2, name: 'Phone', price: 699 },
    { id: 3, name: 'Tablet', price: 499 }
];

// Find product and get just the name
const productName = products.find(p => p.id === 2)?.name;
// 'Phone'

// Find product and transform it
const productCard = products.find(p => p.id === 2);
const formatted = productCard
    ? `${productCard.name}: $${productCard.price}`
    : 'Not found';
// 'Phone: $699'
```

3.6.2 Pattern: Validate Then Process

```
const items = [
    { name: 'A', quantity: 5, price: 10 },
    { name: 'B', quantity: 0, price: 20 },
    { name: 'C', quantity: 3, price: 15 }
];

// Check if all items have valid quantities before calculating
const allValid = items.every(item => item.quantity >= 0);

if (allValid) {
    const total = items
        .filter(item => item.quantity > 0)  // Only items with quantity
        .map(item => item.quantity * item.price)  // Calculate line totals
        .reduce((sum, lineTotal) => sum + lineTotal, 0);  // Sum up

    console.log(`Total: $${total}`);  // Total: $95
}
```

3.6.3 Pattern: Extract from Nested Structure

```
const organization = {
    departments: [
        {
            name: 'Engineering',
            employees: [
                { name: 'Alice', skills: ['JS', 'React'] },
                { name: 'Bob', skills: ['Python', 'JS'] }
            ]
        },
        {
            name: 'Design',
            employees: [
                { name: 'Charlie', skills: ['Figma', 'CSS'] }
            ]
        }
    ]
};

// Find all unique skills in the organization
const allSkills = organization.departments
    .flatMap(dept => dept.employees)
    .flatMap(emp => emp.skills);

const uniqueSkills = [...new Set(allSkills)];
// ['JS', 'React', 'Python', 'Figma', 'CSS']

// Find if anyone knows React
const hasReactDev = organization.departments
    .flatMap(dept => dept.employees)
    .some(emp => emp.skills.includes('React'));
// true
```

3.7 Decision Guide: Which Method to Use?

```
Do you need...
+- All items that match? | - filter() | +- Just the first match? | - find() | +-
The position of first match? | - findIndex() | +- To know if ANY item matches? |
- some() | +- To know if ALL items match? | - every() | +- To transform each item?
| +- One-to-one transformation? | | - map() | - One-to-many (nested results)? | -
flatMap() | - To combine into single value? - reduce()
```

3.8 Practice Exercises

Exercise 3.1: User Lookup

Implement these functions using the appropriate methods:

Exercise

```javascript
const users = [
    { id: 1, name: 'Alice', role: 'admin', active: true },
    { id: 2, name: 'Bob', role: 'user', active: false },
    { id: 3, name: 'Charlie', role: 'user', active: true },
    { id: 4, name: 'Diana', role: 'admin', active: true }
];

// 1. Find user by ID
const findById = id => users.find(u => u.id === id);

// 2. Check if any user is inactive
const hasInactiveUsers = () => users.some(u => !u.active);

// 3. Check if all admins are active
const allAdminsActive = () => users
    .filter(u => u.role === 'admin')
    .every(u => u.active);

// 4. Get index of first inactive user
const findFirstInactiveIndex = () => users.findIndex(u => !u.active);
```

Exercise 3.2: Nested Data Extraction

Given this data structure, extract all product names across all categories:

Exercise

```javascript
const store = {
    categories: [
        {
            name: 'Electronics',
            products: [
                { name: 'Laptop', price: 999 },
                { name: 'Phone', price: 699 }
            ]
        },
        {
            name: 'Clothing',
            products: [
                { name: 'Shirt', price: 29 },
                { name: 'Pants', price: 49 },
                { name: 'Hat', price: 19 }
            ]
        }
    ]
};

// Solution:
const productNames = store.categories
    .flatMap(cat => cat.products)
    .map(p => p.name);
// ['Laptop', 'Phone', 'Shirt', 'Pants', 'Hat']
```

Exercise 3.3: Shopping Cart Operations

Implement these cart operations:

Exercise

```javascript
const cart = [
    { id: 1, name: 'Laptop', price: 999, quantity: 1 },
    { id: 2, name: 'Mouse', price: 29, quantity: 2 },
    { id: 3, name: 'Keyboard', price: 79, quantity: 1 },
    { id: 4, name: 'Monitor', price: 299, quantity: 0 }  // Out of stock
];

// 1. Find item by ID
const findItem = id => cart.find(item => item.id === id);

// 2. Check if cart has any out-of-stock items
const hasOutOfStock = () => cart.some(item => item.quantity === 0);

// 3. Check if all items are available
const allAvailable = () => cart.every(item => item.quantity > 0);

// 4. Get available items summary
const getAvailableItemsSummary = () => cart
    .filter(item => item.quantity > 0)
    .map(item => `${item.name} (x${item.quantity})`);
// ['Laptop (x1)', 'Mouse (x2)', 'Keyboard (x1)']
```

3.9 Chapter Summary

Chapter Summary

Method	Returns	Stops Early?	Use Case
`find`	First match or `undefined`	Yes (first match)	Get single item by condition
`findIndex`	Index or `-1`	Yes (first match)	Get position for updates
`some`	`true` / `false`	Yes (first `true`)	Check if any item matches
`every`	`true` / `false`	Yes (first `false`)	Validate all items
`flatMap`	Flattened array	No	Transform + flatten nested results

Key Insight

`find`, `some`, and `every` are optimized for early exit—use them instead of `filter` when you only need to know about existence or validity.

Part I Summary: Building Blocks Mastered

You now have the foundation:

Concept	What You Learned
First-Class Functions	Functions are values—store, pass, return them
Higher-Order Functions	Functions that take/return functions
Closures	Inner functions remember outer scope
`map`	Transform every element (1:1)
`filter`	Keep elements that pass test
`reduce`	Accumulate to single value
`find` / `findIndex`	Get first match (with early exit)
`some` / `every`	Boolean checks (with early exit)
`flatMap`	Transform and flatten nested results
Method Chaining	Combine operations in readable pipelines

What's Next

In **Part II: Patterns & Techniques**, you'll learn:

- Creating powerful function factories (Chapter 4)
- Building data pipelines with `pipe` and `compose` (Chapter 5)
- Partial application and currying (Chapter 6)
- The systematic protocol for converting imperative code (Chapter 7)

Part I Practice Project: User Dashboard Data Pipeline

Practice Project: User Dashboard Data Pipeline

Apply everything from Part I to build a complete data processing solution:

```
// Raw data from API
const rawData = {
    users: [
        { id: 1, name: 'Alice Johnson', email: 'ALICE@COMPANY.COM',
          department: 'Engineering', salary: 95000, active: true,
          startDate: '2020-03-15' },
        { id: 2, name: 'Bob Smith', email: 'bob@company.com',
          department: 'Sales', salary: 75000, active: true,
          startDate: '2019-07-22' },
        { id: 3, name: 'Charlie Brown', email: 'CHARLIE@COMPANY.COM',
          department: 'Engineering', salary: 105000, active: false,
          startDate: '2018-01-10' },
        { id: 4, name: 'Diana Ross', email: 'diana@company.com',
          department: 'Engineering', salary: 115000, active: true,
          startDate: '2017-11-30' },
        { id: 5, name: 'Eve Wilson', email: 'eve@company.com',
          department: 'Sales', salary: 80000, active: true,
          startDate: '2021-02-14' }
    ],
    departments: [
        { name: 'Engineering', budget: 500000 },
        { name: 'Sales', budget: 300000 }
    ]
};
```

Tasks:

1. **Get all active users with normalized emails (lowercase)**
2. **Calculate total salary expense for active employees**
3. **Group active users by department**
4. **Find the highest paid active employee**
5. **Check if all Engineering employees are active**
6. **Get list of departments that have at least one active employee**
7. **Calculate average salary per department (active employees only)**
8. **Create a summary string for each active user**

Solutions:

```
// 1. Active users with normalized emails
const activeUsers = rawData.users
    .filter(u => u.active)
    .map(u => ({ ...u, email: u.email.toLowerCase() }));

// 2. Total salary expense
const totalSalary = rawData.users
    .filter(u => u.active)
    .reduce((sum, u) => sum + u.salary, 0);

// 3. Group by department
const byDepartment = rawData.users
    .filter(u => u.active)
    .reduce((acc, u) => {
        if (!acc[u.department]) acc[u.department] = [];
        acc[u.department].push(u);
        return acc;
    }, {});

// 4. Highest paid active employee
const highestPaid = rawData.users
    .filter(u => u.active)
    .reduce((max, u) => u.salary > max.salary ? u : max);

// 5. All Engineering employees active?
const allEngActive = rawData.users
    .filter(u => u.department === 'Engineering')
    .every(u => u.active);

// 6. Departments with active employees
const activeDepts = [...new Set(
    rawData.users
        .filter(u => u.active)
        .map(u => u.department)
)];

// 7. Average salary per department
const avgByDept = rawData.users
    .filter(u => u.active)
    .reduce((acc, u) => {
        if (!acc[u.department]) {
            acc[u.department] = { total: 0, count: 0 };
        }
        acc[u.department].total += u.salary;
        acc[u.department].count += 1;
        return acc;
    }, {});
// Then: Object.entries(avgByDept).map(([k, v]) => [k, v.total / v.count])

// 8. Summary strings
const summaries = rawData.users
    .filter(u => u.active)
    .map(u => `${u.name} (${u.department}) - $${u.salary}`);
```

hofbook

hofcode hofvisual

Higher-Order Functions in JavaScript

Part II: Patterns & Techniques

Your Name

Version 1.0

January 13, 2026

https://github.com/anassk01/higher-order-functions-js-book

Contents

Part II

Patterns & Techniques

Chapter 4

Function Factories & Closures

Function factories are HOFs that create and return new functions. They're the foundation of many powerful patterns in JavaScript—from event handlers to middleware to React hooks.

4.1 The Factory Pattern

A function factory creates specialized functions based on configuration:

```javascript
// Factory: creates greeting functions
function createGreeter(greeting) {
    return function(name) {
        return `${greeting}, ${name}!`;
    };
}

// Create specialized greeters
const sayHello = createGreeter('Hello');
const sayHi = createGreeter('Hi');
const sayHey = createGreeter('Hey');

// Use them
sayHello('Alice');  // 'Hello, Alice!'
sayHi('Bob');       // 'Hi, Bob!'
sayHey('Charlie');  // 'Hey, Charlie!'
```

4.1.1 Why This Matters

Instead of writing repetitive code:

× **Bad**

```javascript
// Without factory: repetitive, coupled code
function greetWithHello(name) { return `Hello, ${name}!`; }
function greetWithHi(name) { return `Hi, ${name}!`; }
function greetWithHey(name) { return `Hey, ${name}!`; }
```

You get configurable, DRY code:

Good

```javascript
// With factory: configurable, DRY
const sayHello = createGreeter('Hello');
const sayHi = createGreeter('Hi');
const sayHey = createGreeter('Hey');
```

4.2 Understanding Closures

Definition: Closure

A **closure** is a function that "remembers" variables from its outer scope, even after that outer function has finished executing.

4.2.1 The Mechanism

```
function outer() {
    const secret = 'I am hidden';  // Local variable

    function inner() {
        console.log(secret);  // Inner function accesses outer's variable
    }

    return inner;  // Return the inner function
}

const myFunction = outer();  // outer() runs and returns inner
// At this point, outer() is done executing
// But...

myFunction();  // 'I am hidden' — inner still has access to secret!
```

The closure "closes over" the variable `secret`, keeping it alive in memory even after `outer()` has returned.

What happened?

1. `outer()` creates `secret` and `inner`
2. `inner` references `secret`
3. `outer()` returns `inner` and finishes
4. Normally, `secret` would be garbage collected
5. But `inner` still needs it, so JavaScript keeps `secret` alive
6. This is the closure—`inner` "closes over" `secret`

4.2.2 Visualizing Closure

```
┌───────────────────────┐ │ outer() Scope │
│ ┌────────────────────┐ │ │ │ secret = ’I am
hidden’ │ │ │ │ │ │ │ │ ┌─────────────────┐ │ │ │ │ │
inner() Scope │ │ │ │ │ │ • Has access to: secret │ │ │ │ │ │ • This
access persists │ │ │ │ │ └─────────────────┘ │ │ │
└──────────────────────┘ │ └───────────────────────┘
After outer() returns: - outer’s scope would normally be garbage collected - But
inner() still references secret - So the binding is preserved (the closure)
```

4.2.3 Each Call Creates a New Closure

```
function createCounter() {
    let count = 0;  // Each call gets its own count
    return function() {
        count++;
        return count;
    };
}

const counterA = createCounter();
const counterB = createCounter();

counterA();  // 1
counterA();  // 2
counterA();  // 3

counterB();  // 1 (separate count!)
counterB();  // 2
```

`counterA` and `counterB` each have their own `count` variable—they don’t share.

4.3 Private State with Closures

Closures let you create truly private data—inaccessible from outside:

Bank Account with Private Balance

```
function createBankAccount(initialBalance) {
    let balance = initialBalance;  // Private!

    return {
        deposit(amount) {
            if (amount > 0) {
                balance += amount;
                return balance;
            }
            throw new Error('Deposit amount must be positive');
        },

        withdraw(amount) {
            if (amount > 0 && amount <= balance) {
                balance -= amount;
                return balance;
            }
            throw new Error('Invalid withdrawal');
        },

        getBalance() {
            return balance;
        }
    };
}

const account = createBankAccount(100);

account.getBalance();   // 100
account.deposit(50);    // 150
account.withdraw(30);   // 120

// Cannot access balance directly!
account.balance;        // undefined
```

Key Insight

This is **real** privacy—not convention (like `_balance`), not symbols, actual inaccessibility. The `balance` variable cannot be accessed from outside the closure.

4.4 Practical Factory Patterns

4.4.1 Pattern 1: Configured API Fetchers

```
function createApiClient(baseUrl, defaultHeaders = {}) {
    return {
        async get(endpoint) {
            const response = await fetch(`${baseUrl}${endpoint}`, {
                method: 'GET',
                headers: defaultHeaders
            });
            return response.json();
        },

        async post(endpoint, data) {
            const response = await fetch(`${baseUrl}${endpoint}`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                    ...defaultHeaders
                },
                body: JSON.stringify(data)
            });
            return response.json();
        }
    };
}

// Create clients for different APIs
const github = createApiClient('https://api.github.com', {
    'Authorization': 'token xxx'
});

const myApi = createApiClient('https://api.myapp.com', {
    'X-API-Key': 'my-key'
});

// Use them
const user = await github.get('/users/octocat');
const products = await myApi.get('/products');
```

4.4.2 Pattern 2: Event Handler Factories

```
function createClickHandler(action, options = {}) {
    const { preventDefault = true, stopPropagation = false, log = false } = options;

    return function(event) {
        if (preventDefault) event.preventDefault();
        if (stopPropagation) event.stopPropagation();
        if (log) console.log(`Action: ${action}`, event);

        // Perform the action
        switch (action) {
            case 'save':
                saveForm();
                break;
            case 'delete':
                deleteItem();
                break;
            case 'toggle':
                toggleState();
                break;
        }
    };
}

// Create handlers
const saveHandler = createClickHandler('save', { log: true });
const deleteHandler = createClickHandler('delete', { stopPropagation: true });

// Attach to elements
saveButton.addEventListener('click', saveHandler);
deleteButton.addEventListener('click', deleteHandler);
```

4.4.3 Pattern 3: Validator Factories

```
function createValidator(rules) {
    return function(value) {
        const errors = [];

        for (const rule of rules) {
            if (!rule.test(value)) {
                errors.push(rule.message);
            }
        }

        return {
            valid: errors.length === 0,
            errors
        };
    };
}

// Create specific validators
const validateEmail = createValidator([
    { test: v => v.length > 0, message: 'Email is required' },
    { test: v => v.includes('@'), message: 'Email must contain @' },
    { test: v => v.includes('.'), message: 'Email must contain a domain' }
]);

const validatePassword = createValidator([
    { test: v => v.length >= 8, message: 'Password must be at least 8 characters' },
    { test: v => /[A-Z]/.test(v), message: 'Password must contain uppercase' },
    { test: v => /[0-9]/.test(v), message: 'Password must contain a number' }
]);

// Use them
validateEmail('test@example.com');  // { valid: true, errors: [] }
validateEmail('invalid');           // { valid: false, errors: [...] }

validatePassword('weak');           // { valid: false, errors: [...] }
validatePassword('Strong1234');     // { valid: true, errors: [] }
```

4.4.4 Pattern 4: Memoization Factory

```
function createMemoized(fn) {
    const cache = new Map();

    return function(...args) {
        const key = JSON.stringify(args);

        if (cache.has(key)) {
            console.log('Cache hit for:', key);
            return cache.get(key);
        }

        console.log('Cache miss for:', key);
        const result = fn(...args);
        cache.set(key, result);
        return result;
    };
}

// Memoized Fibonacci
const fastFib = createMemoized(function(n) {
    if (n <= 1) return n;
    return fastFib(n - 1) + fastFib(n - 2);
});

fastFib(40);  // Fast! Uses cache for repeated calculations
```

4.4.5 Pattern 5: Rate Limiter Factory

```
function createRateLimiter(fn, limit, windowMs) {
    let calls = 0;
    let windowStart = Date.now();

    return async function(...args) {
        const now = Date.now();

        // Reset window if expired
        if (now - windowStart >= windowMs) {
            calls = 0;
            windowStart = now;
        }

        // Check limit
        if (calls >= limit) {
            throw new Error(`Rate limit exceeded. Max ${limit} calls per
              ↪  ${windowMs}ms`);
        }

        calls++;
        return fn(...args);
    };
}

// Allow max 5 API calls per second
const limitedFetch = createRateLimiter(fetch, 5, 1000);

// Use it
try {
    await limitedFetch('/api/data');  // Works
    await limitedFetch('/api/data');  // Works
    // ... after 5 calls in < 1 second:
    await limitedFetch('/api/data');  // Throws!
} catch (e) {
    console.log(e.message);
}
```

4.5 Closure Memory Considerations

Closures keep references alive. This is powerful but can cause memory issues.

4.5.1 The Problem: Unintended Retention

```
function processData() {
    const hugeData = new Array(1000000).fill('x');  // ~8MB

    // We only need a small piece
    const summary = hugeData.length;

    // This closure captures the entire scope
    return function() {
        return summary;  // Only uses summary
    };
}

const getSummary = processData();
// hugeData is STILL in memory because the closure exists!
```

4.5.2 The Fix: Minimize Closure Scope

```
function processData() {
    let summary;

    {
        // Block scope limits what the closure can capture
        const hugeData = new Array(1000000).fill('x');
        summary = hugeData.length;
        // hugeData goes out of scope here
    }

    return function() {
        return summary;  // Only captures summary
    };
}

// Or explicitly nullify
function processData() {
    let hugeData = new Array(1000000).fill('x');
    const summary = hugeData.length;

    hugeData = null;  // Release the reference

    return function() {
        return summary;
    };
}
```

4.5.3 The Shared Context Trap

All closures in the same scope share one context object:

```
function createHandlers() {
    const sharedData = { /* large object */ };

    // BOTH closures keep sharedData alive
    const handler1 = () => console.log('handler1');  // Doesn't use sharedData
    const handler2 = () => console.log(sharedData);  // Uses sharedData

    return { handler1, handler2 };
}

const { handler1 } = createHandlers();
// Even though we only kept handler1, sharedData is retained
// because handler1's closure context includes it
```

> Closures keep references alive. Be careful not to accidentally retain large objects in memory.

4.5.4 Best Practice: Factory Function Hygiene

× Bad

```
// BAD: Accidentally capturing large data
function createLoggerBad(config) {
    // config might be huge, but we only need prefix
    return (message) => console.log(`[${config.prefix}] ${message}`);
}
```

Good

```
// GOOD: Only close over what you need
function createLogger(prefix) {
    // prefix is small, intentionally captured
    return (message) => console.log(`[${prefix}] ${message}`);
}

// Or extract what you need
function createLoggerFixed(config) {
    const prefix = config.prefix;  // Extract only what's needed
    return (message) => console.log(`[${prefix}] ${message}`);
}
```

4.6 The Module Pattern

Closures enable the classic module pattern—private state with public interface:

```
const UserModule = (function() {
    // Private
    let users = [];
    let nextId = 1;

    function generateId() {
        return nextId++;
    }
    // Public API (returned object)
    return {
        add(name) {
            const user = { id: generateId(), name };
            users.push(user);
            return user;
        },

        remove(id) {
            users = users.filter(u => u.id !== id);
        },

        getAll() {
            return [...users];  // Return copy to prevent mutation
        },

        findById(id) {
            return users.find(u => u.id === id);
        }
    };
})();  // Immediately invoked!

// Usage
UserModule.add('Alice');      // { id: 1, name: 'Alice' }
UserModule.add('Bob');        // { id: 2, name: 'Bob' }
UserModule.getAll();          // [{ id: 1, ... }, { id: 2, ... }]

// Cannot access internals
UserModule.users;             // undefined
UserModule.nextId;            // undefined
UserModule.generateId;        // undefined
```

4.7 Common Closure Pitfalls

4.7.1 Pitfall 1: Loop Variables

× **Bad**

```
// BROKEN: All handlers log 5
for (var i = 0; i < 5; i++) {
    buttons[i].addEventListener('click', function() {
        console.log(i);  // Always 5!
    });
}
```

Use `let` instead of `var` to create block-scoped variables that work correctly with closures in loops.

Good

```javascript
// FIX 1: Use let (block-scoped)
for (let i = 0; i < 5; i++) {
    buttons[i].addEventListener('click', function() {
        console.log(i);  // Correct: 0, 1, 2, 3, 4
    });
}

// FIX 2: Create new scope with factory
for (var i = 0; i < 5; i++) {
    buttons[i].addEventListener('click', createHandler(i));
}

function createHandler(index) {
    return function() {
        console.log(index);  // Each has its own index
    };
}

// FIX 3: IIFE (Immediately Invoked Function Expression)
for (var i = 0; i < 5; i++) {
    (function(index) {
        buttons[index].addEventListener('click', function() {
            console.log(index);
        });
    })(i);
}
```

4.7.2 Pitfall 2: this Binding in Closures

```javascript
const obj = {
    name: 'MyObject',

    // BROKEN: Regular function loses this
    delayedLog() {
        setTimeout(function() {
            console.log(this.name);  // undefined! this is window/global
        }, 1000);
    },

    // FIX 1: Arrow function (inherits this)
    delayedLogArrow() {
        setTimeout(() => {
            console.log(this.name);  // 'MyObject'
        }, 1000);
    },

    // FIX 2: Save this reference
    delayedLogSaved() {
        const self = this;
        setTimeout(function() {
            console.log(self.name);  // 'MyObject'
        }, 1000);
    },
    // FIX 3: Bind
    delayedLogBound() {
        setTimeout(function() {
            console.log(this.name);  // 'MyObject'
        }.bind(this), 1000);
    }
};
```

4.7.3 Pitfall 3: Stale Closures (React)

```
// React example - common bug
function Counter() {
    const [count, setCount] = useState(0);

    useEffect(() => {
        const interval = setInterval(() => {
            console.log(count);  // Always logs 0!
            setCount(count + 1); // Always sets to 1!
        }, 1000);

        return () => clearInterval(interval);
    }, []);  // Empty deps = closure captures initial count (0)

    return <div>{count}</div>;
}

// FIX: Use functional update
function CounterFixed() {
    const [count, setCount] = useState(0);

    useEffect(() => {
        const interval = setInterval(() => {
            setCount(c => c + 1);  // c is always current
        }, 1000);

        return () => clearInterval(interval);
    }, []);

    return <div>{count}</div>;
}
```

4.8 Chapter Summary

Chapter Summary

Pattern	Description
Function Factory	A function that creates and returns other functions
Closure	A function that retains access to its outer scope
Private State	Variables inaccessible from outside, protected by closure
Module Pattern	IIFE that returns public API, hides private implementation
Closure Scope	All closures in same function share one context object

Key Insight

Closures let you create functions with “memory”—they remember configuration, accumulate state, and encapsulate private data.

4.9 Practice Exercises

Exercise 4.1: Counter Factory

Create a `createCounter` factory that returns an object with:

- `increment()` – increases count by 1, returns new count
- `decrement()` – decreases count by 1, returns new count
- `reset()` – resets to initial value, returns it
- `getCount()` – returns current count

Exercise

```
function createCounter(initialValue = 0) {
    // Your code here
}

// Usage:
const counter = createCounter(10);
counter.increment();  // 11
counter.increment();  // 12
counter.decrement();  // 11
counter.getCount();   // 11
counter.reset();      // 10
```

Exercise 4.2: Once Factory

Create a once factory that ensures a function can only be called once:

Exercise

```
function once(fn) {
    // Your code here
}

// Usage:
const initialize = once(() => {
    console.log('Initializing...');
    return 'initialized';
});

initialize();  // Logs 'Initializing...', returns 'initialized'
initialize();  // Does nothing, returns 'initialized' (cached)
initialize();  // Does nothing, returns 'initialized' (cached)
```

Exercise 4.3: Debounce Factory

Create a debounce factory that delays function execution until after a wait period of inactivity:

Exercise

```
function debounce(fn, waitMs) {
    // Your code here
}

// Usage:
const debouncedSearch = debounce((query) => {
    console.log('Searching for:', query);
}, 300);

// Rapid calls
debouncedSearch('h');
debouncedSearch('he');
debouncedSearch('hel');
debouncedSearch('hell');
debouncedSearch('hello');
// Only logs: 'Searching for: hello' (after 300ms of no calls)
```

Exercise 4.4: Private Stack

Create a stack data structure using closures with true private data:

Exercise

```
function createStack() {
    // Your code here
}

// Usage:
const stack = createStack();
stack.push(1);
stack.push(2);
stack.push(3);
stack.peek();    // 3 (doesn't remove)
stack.pop();     // 3
stack.pop();     // 2
stack.size();    // 1
stack.isEmpty(); // false

// These should not work:
stack.items;     // undefined (private!)
```

Chapter 5

Composition & Pipelines

Composition is the art of combining simple functions to build complex behavior. Instead of one large function, you create small, focused functions and connect them.

5.1 The Composition Concept

5.1.1 Mathematical Composition

In math, $f \circ g$ means “apply g, then apply f”:

$$(f \circ g)(x) = f(g(x))$$

Read right-to-left: first g, then f.

5.1.2 In JavaScript

```javascript
// Two simple functions
const addOne = x => x + 1;
const double = x => x * 2;

// Manual composition
const addOneThenDouble = x => double(addOne(x));

addOneThenDouble(5);  // double(addOne(5)) = double(6) = 12
```

5.1.3 The Problem with Nesting

As you add more functions, nesting becomes unreadable:

```javascript
// Hard to read: inside-out
const result = format(validate(parse(sanitize(input))));

// What's the order? You have to read inside-out:
// 1. sanitize
// 2. parse
// 3. validate
// 4. format
```

5.2 compose: Right-to-Left Composition

`compose` creates a function that applies functions right-to-left:

```
// compose implementation
const compose = (...fns) => x => fns.reduceRight((acc, fn) => fn(acc), x);

// Usage
const addOne = x => x + 1;
const double = x => x * 2;
const square = x => x * x;

const computed = compose(square, double, addOne);
// Reads: square after double after addOne
// Executes: addOne -> double -> square

computed(5);  // square(double(addOne(5))) = square(double(6)) = square(12) = 144
```

5.2.1 How compose Works

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

// Let's trace compose(square, double, addOne)(5):

// fns = [square, double, addOne]
// x = 5

// reduceRight starts from the RIGHT:
// Step 1: acc=5, fn=addOne -> addOne(5) = 6
// Step 2: acc=6, fn=double -> double(6) = 12
// Step 3: acc=12, fn=square -> square(12) = 144

// Result: 144
```

`compose` follows mathematical convention—function composition reads right-to-left. This is preferred in academic functional programming.

5.3 pipe: Left-to-Right Composition

`pipe` is the more intuitive version—functions apply left-to-right:

```
// pipe implementation
const pipe = (...fns) => x => fns.reduce((acc, fn) => fn(acc), x);

// Usage
const addOne = x => x + 1;
const double = x => x * 2;
const square = x => x * x;

const computed = pipe(addOne, double, square);
// Reads: addOne then double then square
// Executes: addOne -> double -> square
```

```
computed(5);  // 144 (same result, but reads naturally)
```

5.3.1 pipe is the Web Dev Standard

Most JavaScript libraries and frameworks prefer `pipe`:

- RxJS uses `pipe()`
- fp-ts uses `pipe()`
- Effect-TS uses `pipe()`
- Redux middleware flows left-to-right

Why? It reads like English: “Do this, then this, then this.”

```
// pipe reads top-to-bottom, left-to-right
const processUser = pipe(
    validateInput,
    normalizeEmail,
    hashPassword,
    saveToDatabase
);

// compose reads bottom-to-top, right-to-left
const processUser = compose(
    saveToDatabase,
    hashPassword,
    normalizeEmail,
    validateInput
);
```

5.4 Building Pipelines

5.4.1 Data Transformation Pipeline

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

// Small, focused functions
const trim = str => str.trim();
const toLowerCase = str => str.toLowerCase();
const replaceSpaces = str => str.replace(/\s+/g, '-');
const removeSpecialChars = str => str.replace(/[^a-z0-9-]/g, '');

// Combine into pipeline
const slugify = pipe(
    trim,
    toLowerCase,
    replaceSpaces,
    removeSpecialChars
);

slugify('  Hello World! This is a TEST  ');
// 'hello-world-this-is-a-test'
```

5.4.2 Request Processing Pipeline

```javascript
// Each function transforms the request object
const addTimestamp = req => ({
    ...req,
    timestamp: Date.now()
});

const validateAuth = req => {
    if (!req.headers.authorization) {
        throw new Error('Unauthorized');
    }
    return req;
};

const parseBody = req => ({
    ...req,
    body: JSON.parse(req.rawBody)
});

const sanitizeInput = req => ({
    ...req,
    body: {
        ...req.body,
        email: req.body.email?.toLowerCase().trim()
    }
});

// Pipeline
const processRequest = pipe(
    addTimestamp,
    validateAuth,
    parseBody,
    sanitizeInput
);

// Usage
const result = processRequest({
    headers: { authorization: 'Bearer xxx' },
    rawBody: '{”email”: ” ALICE@TEST.COM ”}'
});
```

5.4.3 Array Processing Pipeline

```javascript
const users = [
    { name: 'Alice', age: 25, active: true },
    { name: 'Bob', age: 17, active: true },
    { name: 'Charlie', age: 30, active: false },
    { name: 'Diana', age: 22, active: true }
];

// Functions that work on arrays
const filterActive = users => users.filter(u => u.active);
const filterAdults = users => users.filter(u => u.age >= 18);
const sortByAge = users => [...users].sort((a, b) => a.age - b.age);
const extractNames = users => users.map(u => u.name);

// Pipeline
const getActiveAdultNames = pipe(
    filterActive,
    filterAdults,
    sortByAge,
    extractNames
);

getActiveAdultNames(users);  // ['Diana', 'Alice']
```

5.5 The tap Utility: Debugging Pipelines

`tap` lets you peek at values without affecting the pipeline:

```javascript
const tap = fn => value => {
    fn(value);
    return value;  // Pass through unchanged
};

// Usage: debug a pipeline
const processData = pipe(
    filterActive,
    tap(data => console.log('After filter:', data.length)),
    sortByAge,
    tap(data => console.log('After sort:', data)),
    extractNames,
    tap(names => console.log('Final:', names))
);
```

5.5.1 tap Variations

```javascript
// Log with label
const tapLog = label => tap(x => console.log(label, x));

// Debugger breakpoint
const tapDebug = tap(() => debugger);

// Conditional tap
const tapIf = (predicate, fn) => value => {
    if (predicate(value)) fn(value);
    return value;
};

// Usage
const processData = pipe(
    filterActive,
    tapLog('After filter:'),
    tapIf(arr => arr.length === 0, () => console.warn('No active users!')),
    sortByAge,
    extractNames
);
```

5.6 Point-Free Style

Point-free (or tacit) style means defining functions without explicitly mentioning their arguments:

```
// Pointed (explicit argument)
const getLength = str => str.length;
const isEven = n => n % 2 === 0;

// Point-free
const getLength = str => str.length;  // Can't really avoid the arg here

// But with composition:
// Pointed
const getLengthAndDouble = str => double(str.length);

// Point-free (no mention of str)
const getLengthAndDouble = pipe(
    str => str.length,
    double
);
```

5.6.1 Point-Free with Higher-Order Utilities

```
// Helper to get property
const prop = key => obj => obj[key];

// Helper to call method
const method = (name, ...args) => obj => obj[name](...args);

// Now we can go point-free
const getName = prop('name');
const getEmail = prop('email');
const toUpperCase = method('toUpperCase');

// Point-free pipeline
const getUpperName = pipe(
    prop('name'),
    method('toUpperCase')
);

getUpperName({ name: 'alice', email: 'a@b.com' });  // 'ALICE'
```

5.6.2 When to Use Point-Free

Good: Simple, linear transformations

Good

```
// Clear and readable
const processName = pipe(trim, toLowerCase, capitalize);
```

Bad: Complex logic or when it hurts readability

× **Bad**

```
// This is too clever
const process = pipe(
    fork(join, head, tail),
    converge(multiply, [add(1), subtract(1)])
);
```

Good

```
// Just write it clearly
const process = x => {
    const a = x + 1;
    const b = x - 1;
    return a * b;
};
```

Guideline: If you have to think hard to understand it, don't use point-free.

5.7 Composing with Multiple Arguments

Basic `pipe` and `compose` work with single-argument functions. For multiple arguments, we need additional techniques (covered in Chapter 6: Currying).

For now, here's a multi-argument compose:

```
// First function can take multiple args, rest take one
const pipeWith = (...fns) => (...args) => {
    const [first, ...rest] = fns;
    return rest.reduce((acc, fn) => fn(acc), first(...args));
};

// Usage
const add = (a, b) => a + b;
const double = x => x * 2;
const square = x => x * x;

const compute = pipeWith(add, double, square);
compute(2, 3);  // square(double(add(2, 3))) = square(double(5)) = square(10) = 100
```

5.8 Real-World Composition Patterns

5.8.1 Pattern 1: Validation Pipeline

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

// Each validator returns { valid, value, error }
const createValidator = (test, errorMsg) => result => {
    if (!result.valid) return result;  // Short-circuit on first error

    if (!test(result.value)) {
        return { valid: false, value: result.value, error: errorMsg };
    }
    return result;
};

const notEmpty = createValidator(
    v => v.trim().length > 0,
    'Cannot be empty'
);

const minLength = n => createValidator(
    v => v.length >= n,
    `Must be at least ${n} characters`
);

const hasUppercase = createValidator(
    v => /[A-Z]/.test(v),
    'Must contain uppercase letter'
);

const hasNumber = createValidator(
    v => /[0-9]/.test(v),
    'Must contain a number'
);

// Compose validators
const validatePassword = pipe(
    notEmpty,
    minLength(8),
    hasUppercase,
    hasNumber
);

// Usage
const result = validatePassword({ valid: true, value: 'weak' });
// { valid: false, value: 'weak', error: 'Must be at least 8 characters' }

const result2 = validatePassword({ valid: true, value: 'StrongPass1' });
// { valid: true, value: 'StrongPass1', error: undefined }
```

5.8.2 Pattern 2: Async Pipeline

```
// Async pipe: awaits each step
const pipeAsync = (...fns) => async x => {
    let result = x;
    for (const fn of fns) {
        result = await fn(result);
    }
    return result;
};

// Usage
```

```
const processUser = pipeAsync(
    fetchUser,
    validateUser,
    enrichWithProfile,
    saveToCache
);

const user = await processUser(userId);
```

5.9 Chapter Summary

Chapter Summary

Function	Direction	Use Case
`compose`	Right-to-left (→)	Mathematical convention, store enhancers
`pipe`	Left-to-right (←)	Web dev standard, data pipelines
`tap`	Pass-through	Debugging, logging
`flow`	Left-to-right	Same as pipe (Lodash name)

Key Insight

Break complex transformations into small, focused functions. Combine them with `pipe` for readable, maintainable code.

5.10 Practice Exercises

Exercise 5.1: Build pipe and compose

Implement both from scratch:

Exercise

```
function pipe(...fns) {
    // Your code here
}

function compose(...fns) {
    // Your code here
}

// Test
const add1 = x => x + 1;
const mult2 = x => x * 2;
const sub3 = x => x - 3;

pipe(add1, mult2, sub3)(5);     // ((5 + 1) * 2) - 3 = 9
compose(sub3, mult2, add1)(5);  // Same result: 9
```

Exercise 5.2: String Processing Pipeline

Create a text processing pipeline that:

1. Trims whitespace
2. Converts to lowercase
3. Removes punctuation
4. Splits into words
5. Removes words shorter than 3 characters
6. Joins with hyphens

Exercise

```
const processText = pipe(
    // Your functions here
);

processText('  Hello, World! This is a TEST...  ');
// 'hello-world-this-test'
```

Chapter 6

Partial Application & Currying

Partial application and currying let you create specialized functions from general ones by pre-filling arguments.

6.1 The Problem: Repeated Arguments

```
// You keep passing the same first argument
fetchFromApi('https://api.github.com', '/users');
fetchFromApi('https://api.github.com', '/repos');
fetchFromApi('https://api.github.com', '/gists');

// Or the same configuration
formatDate(date1, 'YYYY-MM-DD', 'en-US');
formatDate(date2, 'YYYY-MM-DD', 'en-US');
formatDate(date3, 'YYYY-MM-DD', 'en-US');
```

What if you could create a specialized version with some arguments "locked in"?

6.2 Partial Application

Partial application creates a new function with some arguments pre-filled:

```
// General function
function greet(greeting, name) {
    return `${greeting}, ${name}!`;
}

// Partially apply the first argument
function sayHello(name) {
    return greet('Hello', name);
}

function sayHi(name) {
    return greet('Hi', name);
}

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

6.2.1 Using bind for Partial Application

JavaScript's `bind` can partially apply arguments:

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

// bind(thisArg, ...args) - we use null for thisArg
const sayHello = greet.bind(null, 'Hello');
const sayGoodbye = greet.bind(null, 'Goodbye');

sayHello('Alice');    // 'Hello, Alice!'
sayGoodbye('Bob');    // 'Goodbye, Bob!'
```

6.2.2 A Generic partial Function

```javascript
const partial = (fn, ...presetArgs) => {
    return (...laterArgs) => fn(...presetArgs, ...laterArgs);
};

// Usage
function createUser(role, department, name, email) {
    return { role, department, name, email };
}

// Create specialized functions
const createAdmin = partial(createUser, 'admin', 'IT');
const createEmployee = partial(createUser, 'employee', 'Sales');

createAdmin('Alice', 'alice@test.com');
// { role: 'admin', department: 'IT', name: 'Alice', email: 'alice@test.com' }

createEmployee('Bob', 'bob@test.com');
// { role: 'employee', department: 'Sales', name: 'Bob', email: 'bob@test.com' }
```

6.2.3 Web Dev Example: API Client

```javascript
const partial = (fn, ...presetArgs) => (...laterArgs) => fn(...presetArgs, ...laterArgs);

async function apiRequest(baseUrl, method, endpoint, data = null) {
    const options = {
        method,
        headers: { 'Content-Type': 'application/json' }
    };

    if (data) options.body = JSON.stringify(data);

    const response = await fetch(`${baseUrl}${endpoint}`, options);
    return response.json();
}

// Create specialized functions
const githubApi = partial(apiRequest, 'https://api.github.com');
const githubGet = partial(githubApi, 'GET');
const githubPost = partial(githubApi, 'POST');
```

```javascript
// Use them
const user = await githubGet('/users/octocat');
const gist = await githubPost('/gists', { files: {...} });
```

6.3 Currying

Currying transforms a function that takes multiple arguments into a sequence of functions that each take a single argument:

```javascript
// Regular function
function add(a, b, c) {
    return a + b + c;
}
add(1, 2, 3);  // 6

// Curried version
function addCurried(a) {
    return function(b) {
        return function(c) {
            return a + b + c;
        };
    };
}
addCurried(1)(2)(3);  // 6
```

6.3.1 Arrow Function Syntax

```javascript
// Same curried function with arrows
const addCurried = a => b => c => a + b + c;

addCurried(1)(2)(3);  // 6

// Can call incrementally
const add1 = addCurried(1);       // b => c => 1 + b + c
const add1and2 = add1(2);         // c => 1 + 2 + c
const result = add1and2(3);       // 6
```

6.3.2 The Difference: Partial vs Curry

Partial Application	Currying
Fix some arguments at once	Transform to single-arg chain
`partial(fn, a, b)(c, d)`	`curry(fn)(a)(b)(c)(d)`
Flexible grouping	One arg at a time

```javascript
// Partial: fix multiple args at once
const partial = (fn, ...args) => (...more) => fn(...args, ...more);
const add5and6 = partial(add, 5, 6);  // Fix two args
add5and6(7);  // 18

// Curry: always one at a time
const addCurried = a => b => c => a + b + c;
addCurried(5)(6)(7);  // 18
```

6.4 Auto-Curry: Flexible Currying

Auto-curry lets you call with any number of arguments:

```
function curry(fn) {
    return function curried(...args) {
        if (args.length >= fn.length) {
            // Enough arguments: call the function
            return fn(...args);
        } else {
            // Not enough: return function that collects more
            return (...more) => curried(...args, ...more);
        }
    };
}

// Usage
const add = (a, b, c) => a + b + c;
const curriedAdd = curry(add);

// All of these work:
curriedAdd(1, 2, 3);    // 6 (all at once)
curriedAdd(1)(2)(3);    // 6 (one at a time)
curriedAdd(1, 2)(3);    // 6 (mixed)
curriedAdd(1)(2, 3);    // 6 (mixed)
```

6.4.1 How Auto-Curry Works

```
function curry(fn) {
    return function curried(...args) {
        // fn.length is the number of declared parameters
        if (args.length >= fn.length) {
            return fn(...args);  // Call with all args
        }
        return (...more) => curried(...args, ...more);  // Collect more
    };
}

// Trace curry(add)(1)(2)(3):
// Call 1: args=[1], 1 < 3, return (...more) => curried(1, ...more)
// Call 2: args=[1,2], 2 < 3, return (...more) => curried(1, 2, ...more)
// Call 3: args=[1,2,3], 3 >= 3, return add(1, 2, 3) = 6
```

6.5 Currying in Practice

6.5.1 Data-Last for Piping

Curried functions work beautifully with `pipe` when data comes last:

```
const curry = fn => function curried(...args) {
    return args.length >= fn.length
        ? fn(...args)
        : (...more) => curried(...args, ...more);
};

// Curried utilities (data-last)
const map = curry((fn, array) => array.map(fn));
const filter = curry((predicate, array) => array.filter(predicate));
const reduce = curry((fn, initial, array) => array.reduce(fn, initial));

// Now they compose beautifully
const pipe = (...fns) => x => fns.reduce((acc, fn) => fn(acc), x);

const processNumbers = pipe(
    filter(n => n > 0),           // Keep positives
    map(n => n * 2),              // Double them
    reduce((sum, n) => sum + n, 0) // Sum them
);

processNumbers([-1, 2, -3, 4, 5]);  // (2 + 4 + 5) * 2 = 22
```

6.5.2 Web Dev Example: Validation

```
const curry = fn => (...args) =>
    args.length >= fn.length ? fn(...args) : curry(fn.bind(null, ...args));

// Validators return { valid, error } or just boolean
const createRule = curry((test, errorMsg, value) => ({
    valid: test(value),
    error: test(value) ? null : errorMsg,
    value
}));

const minLength = n => createRule(
    v => v.length >= n,
    `Must be at least ${n} characters`
);

const maxLength = n => createRule(
    v => v.length <= n,
    `Must be at most ${n} characters`
);

const matchesPattern = (regex, msg) => createRule(
    v => regex.test(v),
    msg
);

// Compose validators
const validateAll = (...validators) => value => {
    for (const validate of validators) {
        const result = validate(value);
        if (!result.valid) return result;
    }
    return { valid: true, error: null, value };
};

const validateUsername = validateAll(
    minLength(3),
    maxLength(20),
    matchesPattern(/^[a-zA-Z0-9_]+$/, 'Only letters, numbers, underscore')
);

validateUsername('ab');  // { valid: false, error: 'Must be at least 3 characters' }
validateUsername('alice_123');  // { valid: true, error: null, value: 'alice_123' }
```

6.6 Partial Application vs Currying: When to Use Each

6.6.1 Use Partial Application When:

- You want to fix several arguments at once
- Working with existing non-curried functions
- The argument order doesn't match your needs

```
// Fix multiple args at once
const logError = partial(console.log, '[ERROR]', new Date().toISOString());
logError('Something went wrong');  // [ERROR] 2024-01-15T10:30:00.000Z Something went
    ↪  wrong

// Working with existing functions
const parseBase10 = partial(parseInt, undefined, 10);
['1', '2', '3'].map(parseBase10);  // [1, 2, 3] not [1, NaN, NaN]
```

6.6.2 Use Currying When:

- Building composable utilities
- Creating point-free pipelines
- Designing an API meant for composition

```
// Composable utilities
const add = curry((a, b) => a + b);
const multiply = curry((a, b) => a * b);

const add5 = add(5);
const times2 = multiply(2);

const transform = pipe(add5, times2);
transform(10);  // (10 + 5) * 2 = 30
```

6.7 Right-to-Left Partial Application

Sometimes you need to fill arguments from the right:

```
const partialRight = (fn, ...presetArgs) => {
    return (...laterArgs) => fn(...laterArgs, ...presetArgs);
};

// Example: parseInt has (string, radix) signature
const parseInt10 = partialRight(parseInt, 10);

['1', '2', '3'].map(parseInt10);  // [1, 2, 3]

// Compare to the problem:
['1', '2', '3'].map(parseInt);    // [1, NaN, NaN] - radix gets index!
```

6.8 Chapter Summary

Chapter Summary

Technique	Definition	Use Case
Partial Application	Pre-fill some arguments	Specialize existing functions
Currying	Transform to single-arg chain	Composable utilities
Auto-Curry	Flexible argument collection	Best of both worlds
Data-Last	Data argument comes last	Enables piping

Key Insight

Currying and partial application let you build specialized tools from general ones. Combined with `pipe`, they enable powerful, readable data transformations.

6.9 Practice Exercises

Exercise 6.1: Implement curry

Exercise

```javascript
function curry(fn) {
    // Your code here
}

// Test
const add = (a, b, c) => a + b + c;
const curried = curry(add);

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

Exercise 6.2: Implement partial and partialRight

Exercise

```javascript
function partial(fn, ...presetArgs) {
    // Your code here
}

function partialRight(fn, ...presetArgs) {
    // Your code here
}

// Test
const greet = (greeting, name, punct) => `${greeting}, ${name}${punct}`;

const sayHello = partial(greet, 'Hello');
console.log(sayHello('Alice', '!'));  // 'Hello, Alice!'

const greetBob = partialRight(greet, 'Bob', '!');
console.log(greetBob('Hi'));  // 'Hi, Bob!'
```

Chapter 7

The Pipeline Injection Protocol

This chapter presents a systematic algorithm for converting imperative code to functional pipelines. Follow these steps to refactor any loop-based code.

7.1 The 4-Phase Protocol

Phase 1: Isolate State

Identify mutable state that accumulates results.

Phase 2: Extract Predicates

Turn `if` conditions into named pure functions.

Phase 3: Decouple Transformations

Turn mutations into pure transformation functions.

Phase 4: Compose Pipeline

Combine using `filter`, `map`, `reduce`.

7.2 Phase 1: Isolate State

Goal: Find the mutable variables that accumulate results.

7.2.1 Pattern Recognition

Look for:

```
let result = [];           // Accumulator array
let total = 0;             // Accumulator number
let found = null;          // Search result
let isValid = true;        // Flag
```

7.2.2 Example

```
// BEFORE: Find the state
function getActiveEmails(users) {
    const emails = [];  // ← STATE: accumulator
    for (let i = 0; i < users.length; i++) {
        if (users[i].isActive) {
            if (users[i].email) {
                emails.push(users[i].email.toLowerCase());
            }
        }
    }
    return emails;
}
```

Identified State: `emails` array is the accumulator.

7.2.3 State Types and Their HOF Equivalents

State Pattern	HOF Replacement
`results.push(item)`	`map` or `filter`
`total += value`	`reduce`
`found = item; break;`	`find`
`isValid = false; break;`	`some` or `every`
`count++`	`filter().length` or `reduce`

7.3 Phase 2: Extract Predicates

Goal: Turn `if` conditions into named boolean functions.

7.3.1 Pattern Recognition

Every `if (condition)` becomes a predicate:

```
// BEFORE
if (users[i].isActive) { ... }
if (users[i].email) { ... }

// AFTER
const isActive = user => user.isActive;
const hasEmail = user => user.email != null;
```

7.3.2 Example Continued

```
// Extract predicates from the conditions
const isActive = user => user.isActive;
const hasEmail = user => user.email != null;

// The loop conditions are now named functions
```

7.3.3 Combining Predicates

```
// Multiple conditions can be combined
const isActiveWithEmail = user => isActive(user) && hasEmail(user);

// Or kept separate for reusability
// filter(isActive).filter(hasEmail)
```

7.3.4 Common Predicate Patterns

```
// Null/undefined checks
const exists = x => x != null;
const hasProperty = prop => obj => obj[prop] != null;

// Comparisons
const isGreaterThan = n => x => x > n;
const isLessThan = n => x => x < n;
const equals = target => x => x === target;

// String checks
const startsWith = prefix => str => str.startsWith(prefix);
const contains = substr => str => str.includes(substr);
const matchesPattern = regex => str => regex.test(str);

// Object checks
const hasRole = role => user => user.role === role;
const isActive = user => user.active === true;
const belongsTo = dept => emp => emp.department === dept;
```

7.4 Phase 3: Decouple Transformations

Goal: Turn mutations into pure functions that return new values.

7.4.1 Pattern Recognition

Look for:

```
item.prop = value;            // Property mutation
items.push(transform(item)); // Transform + push
result = process(item);       // Assignment after processing
```

7.4.2 Example Continued

```javascript
// BEFORE (inside loop)
emails.push(users[i].email.toLowerCase());

// AFTER
const extractEmail = user => user.email;
const normalizeEmail = email => email.toLowerCase();

// Or combined:
const getEmail = user => user.email.toLowerCase();
```

7.4.3 Common Transformation Patterns

```javascript
// Property extraction
const prop = key => obj => obj[key];
const props = (...keys) => obj => keys.map(k => obj[k]);

// Object reshaping
const pick = (...keys) => obj =>
    keys.reduce((acc, k) => ({ ...acc, [k]: obj[k] }), {});

const omit = (...keys) => obj =>
    Object.keys(obj)
        .filter(k => !keys.includes(k))
        .reduce((acc, k) => ({ ...acc, [k]: obj[k] }), {});

// Value transformations
const toUpperCase = str => str.toUpperCase();
const toLowerCase = str => str.toLowerCase();
const trim = str => str.trim();

// Number transformations
const add = n => x => x + n;
const multiply = n => x => x * n;
const clamp = (min, max) => x => Math.max(min, Math.min(max, x));

// Object transformations
const withDefaults = defaults => obj => ({ ...defaults, ...obj });
const rename = (oldKey, newKey) => obj => {
    const { [oldKey]: value, ...rest } = obj;
    return { ...rest, [newKey]: value };
};
```

7.5 Phase 4: Compose Pipeline

Goal: Combine predicates and transformations into a pipeline.

7.5.1 The Mapping

Imperative	Functional
`if (condition)` inside loop	`.filter(predicate)`
`transform(item)`	`.map(transformer)`
`accumulator += value`	`.reduce(fn, initial)`
`if (...) { break; }`	`.find(predicate)`

7.5.2 Example Completed

× Bad

```
// BEFORE: Imperative
function getActiveEmails(users) {
    const emails = [];
    for (let i = 0; i < users.length; i++) {
        if (users[i].isActive) {
            if (users[i].email) {
                emails.push(users[i].email.toLowerCase());
            }
        }
    }
    return emails;
}
```

Good

```
// AFTER: Functional Pipeline
const isActive = user => user.isActive;
const hasEmail = user => user.email != null;
const getEmail = user => user.email.toLowerCase();

function getActiveEmails(users) {
    return users
        .filter(isActive)
        .filter(hasEmail)
        .map(getEmail);
}

// Or ultra-concise (but less readable):
const getActiveEmails = users => users
    .filter(u => u.isActive && u.email)
    .map(u => u.email.toLowerCase());
```

7.6 Complete Worked Examples

7.6.1 Example 1: Sum of Squares of Evens

× Bad

```
// BEFORE: Imperative
function sumOfSquaresOfEvens(numbers) {
    let sum = 0;  // STATE
    for (let i = 0; i < numbers.length; i++) {
        if (numbers[i] % 2 === 0) {  // PREDICATE
            sum += numbers[i] * numbers[i];  // TRANSFORM + ACCUMULATE
        }
    }
    return sum;
}
```

Good

```
// PHASE 1: State = sum (accumulator) -> reduce
// PHASE 2: Predicate = isEven
// PHASE 3: Transform = square
// PHASE 4: Compose

const isEven = n => n % 2 === 0;
const square = n => n * n;

function sumOfSquaresOfEvens(numbers) {
    return numbers
        .filter(isEven)
        .map(square)
        .reduce((sum, n) => sum + n, 0);
}

// Test
sumOfSquaresOfEvens([1, 2, 3, 4, 5, 6]);  // 4 + 16 + 36 = 56
```

7.6.2 Example 2: Find First Admin

× Bad

```
// BEFORE: Imperative
function findFirstAdmin(users) {
    let admin = null;  // STATE: search result
    for (let i = 0; i < users.length; i++) {
        if (users[i].role === 'admin') {  // PREDICATE
            admin = users[i];  // Found!
            break;  // Early exit
        }
    }
    return admin;
}
```

Good

```
// Early exit + single result -> find

const isAdmin = user => user.role === 'admin';

function findFirstAdmin(users) {
    return users.find(isAdmin);
}
```

7.6.3 Example 3: Validate All Fields

× Bad

```
// BEFORE: Imperative
function validateForm(fields) {
    let isValid = true;  // STATE: flag
    const errors = [];
    for (let i = 0; i < fields.length; i++) {
        if (fields[i].required && !fields[i].value) {  // PREDICATE
            isValid = false;
            errors.push(`${fields[i].name} is required`);
        }
    }
    return { isValid, errors };
}
```

Good

```
// Multiple concerns: validation check + error collection

const isRequiredAndEmpty = field => field.required && !field.value;
const toErrorMessage = field => `${field.name} is required`;

function validateForm(fields) {
    const invalidFields = fields.filter(isRequiredAndEmpty);
    return {
        isValid: invalidFields.length === 0,
        errors: invalidFields.map(toErrorMessage)
    };
}
```

7.6.4 Example 4: Group By Category

× Bad

```
// BEFORE: Imperative
function groupByCategory(products) {
    const groups = {};  // STATE: object accumulator
    for (let i = 0; i < products.length; i++) {
        const category = products[i].category;  // TRANSFORM (extract)
        if (!groups[category]) {
            groups[category] = [];
        }
        groups[category].push(products[i]);  // ACCUMULATE
    }
    return groups;
}
```

Good

```
// Object accumulator → reduce

function groupByCategory(products) {
    return products.reduce((groups, product) => {
        const category = product.category;
        return {
            ...groups,
            [category]: [...(groups[category] || []), product]
        };
    }, {});
}

// More efficient version (mutation in reduce is OK):
function groupByCategory(products) {
    return products.reduce((groups, product) => {
        const category = product.category;
        if (!groups[category]) groups[category] = [];
        groups[category].push(product);
        return groups;
    }, {});
}
```

7.6.5 Example 5: Nested Loop Flattening

× Bad

```
// BEFORE: Imperative (nested loops)
function getAllActiveMembers(teams) {
    const activeMembers = [];
    for (let i = 0; i < teams.length; i++) {
        for (let j = 0; j < teams[i].members.length; j++) {
            if (teams[i].members[j].active) {
                activeMembers.push(teams[i].members[j]);
            }
        }
    }
    return activeMembers;
}
```

Good

```
// Nested loop accessing child array → flatMap

const isActive = member => member.active;
const getMembers = team => team.members;
```

```
function getAllActiveMembers(teams) {
    return teams
        .flatMap(getMembers)
        .filter(isActive);
}
```

7.7 Decision Flowchart

```
START: Analyze the loop │ ├─ Does it build an array? │ ├─ By filtering items?
→ filter() │ ├─ By transforming items? → map() │ └─ By both? → filter().map()
or flatMap() │ ├─ Does it build an object? │ ├─ Key-value lookup? → reduce() to
object │ ├─ Grouping? → reduce() with array values │ └─ Counting? → reduce()
with number values │ ├─ Does it calculate a single value? │ └─ reduce() │ ├─
Does it search for one item? │ ├─ Need the item? → find() │ └─ Need the index?
→ findIndex() │ ├─ Does it check a condition? │ ├─ ”Does any match?” → some() │
└─ ”Do all match?” → every() │ └─ Does it have nested loops? └─ flatMap() for the
inner arrays
```

7.8 Anti-Patterns to Avoid

7.8.1 Anti-Pattern 1: Reduce-Spread (O(n²))

× **Bad**

```
// BAD: Creates new object on every iteration
const byId = items.reduce((acc, item) => ({
    ...acc,
    [item.id]: item
}), {});
```

Good

```
// GOOD: Mutate accumulator (OK in reduce)
const byId = items.reduce((acc, item) => {
    acc[item.id] = item;
    return acc;
}, {});

// ALSO GOOD: Use Object.fromEntries
const byId = Object.fromEntries(
    items.map(item => [item.id, item])
);
```

7.8.2 Anti-Pattern 2: Filter Then Length

× Bad

```javascript
// BAD: Creates intermediate array just to count
const count = items.filter(x => x.active).length;
```

Good

```javascript
// GOOD: Use reduce to count directly
const count = items.reduce((n, x) => x.active ? n + 1 : n, 0);

// ALSO GOOD: If you need the items too, filter is fine
const activeItems = items.filter(x => x.active);
const count = activeItems.length;
```

7.8.3 Anti-Pattern 3: Map for Side Effects

× Bad

```javascript
// BAD: Using map for side effects (returns unused array)
users.map(user => {
    sendEmail(user);  // Side effect!
});
```

Good

```javascript
// GOOD: Use forEach for side effects
users.forEach(user => {
    sendEmail(user);
});

// ALSO GOOD: for...of loop
for (const user of users) {
    sendEmail(user);
}
```

7.9 When NOT to Convert

Sometimes imperative is better:

7.9.1 1. Performance-Critical Code

```javascript
// Array methods have overhead. For millions of items:
// Imperative
let sum = 0;
for (let i = 0; i < hugeArray.length; i++) {
    sum += hugeArray[i];
}
```

```
// May be faster than:
const sum = hugeArray.reduce((a, b) => a + b, 0);
```

7.9.2 2. Complex Control Flow

```
// Hard to express functionally
for (let i = 0; i < items.length; i++) {
    if (condition1) {
        doA();
        continue;
    }
    if (condition2) {
        doB();
        if (condition3) break;
    }
    doC();
}
```

7.9.3 3. Early Exit on Find

```
// find() is good, but for complex exit logic:
for (const item of items) {
    const result = complexOperation(item);
    if (result.success) {
        return result;  // Complex early return
    }
    if (result.fatal) {
        throw new Error(result.message);
    }
}
```

7.9.4 4. Mutation is Intentional

```
// When you actually need to modify in place
for (const item of items) {
    item.processed = true;
    item.timestamp = Date.now();
}
```

7.10 Chapter Summary

Chapter Summary

7.10.1 The 4-Phase Protocol

Phase	Action	Look For
1	Isolate State	`let result = []`, `let sum = 0`
2	Extract Predicates	`if (condition)`
3	Decouple Transforms	`item.x = y`, `push(transform(x))`
4	Compose Pipeline	`filter`, `map`, `reduce`

7.10.2 Quick Reference

Pattern	Description
Loop + if + push	`filter`
Loop + transform + push	`map`
Loop + accumulate value	`reduce`
Nested loops	`flatMap`
Loop + break on find	`find` / `findIndex`
Loop + flag check	`some` / `every`

Key Insight

Most loops follow predictable patterns. Recognize the pattern, apply the corresponding HOF, and your code becomes more declarative and maintainable.

7.11 Practice Exercises

Exercise 7.1: Refactor to Pipeline

Convert this imperative code:

Exercise

```
function processOrders(orders) {
    const result = [];
    for (let i = 0; i < orders.length; i++) {
        if (orders[i].status === 'completed') {
            if (orders[i].total > 100) {
                result.push({
                    id: orders[i].id,
                    total: orders[i].total,
                    discountedTotal: orders[i].total * 0.9
                });
            }
        }
    }
    return result;
}
```

Exercise 7.2: Nested Loop Refactoring

Convert this:

Exercise

```
function getSkillsFromTeams(teams) {
    const skills = [];
    for (let i = 0; i < teams.length; i++) {
        for (let j = 0; j < teams[i].members.length; j++) {
            for (let k = 0; k < teams[i].members[j].skills.length; k++) {
                if (!skills.includes(teams[i].members[j].skills[k])) {
                    skills.push(teams[i].members[j].skills[k]);
                }
            }
        }
    }
    return skills;
}
```

Part II Summary

You now have intermediate techniques:

Concept	What You Learned
Function Factories	Create specialized functions from configuration
Closures	Functions that remember their scope
Private State	True encapsulation via closures
compose/pipe	Combine functions into pipelines
tap	Debug without breaking the pipeline
Partial Application	Pre-fill some arguments
Currying	Transform to single-argument chain
Pipeline Protocol	Systematic imperative-to-functional conversion

What's Next

In **Part III: Real-World Web Development**, you'll apply these patterns to:

- Async JavaScript and Promises (Chapter 8)
- React hooks and components (Chapter 9)
- State management with Redux (Chapter 10)
- Node.js middleware and APIs (Chapter 11)