JavaScript Intermediate

A Complete 7-Chapter Course

Topics covered:

Object.keys/values/entries & spread/rest · Closures · ES6 Classes

this/call/apply/bind · Error Handling · Copying & Mutation · localStorage & JSON

Exercises: 21 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · Quick-reference tables

Table of Contents

CHAPTER 1 OF 7

Object.keys/values/entries and the Spread/Rest Operators

COURSE 2 · CH 1

Object.keys/values/entries and the Spread/Rest Operators

Turning objects into arrays you can already work with, and two compact syntaxes for copying and collecting values

Fundamentals Chapter 7 flagged that `Object.keys()` turns an object's property names into a real array — meaning Chapter 6's `map` / `filter` / `reduce` can be used on object data too. This chapter covers that properly, plus the spread (`...`) and rest (`...`) operators, which look identical but do opposite jobs depending on where they appear.

Object.keys, Object.values, Object.entries

```
const scores = { maths: 88, science: 72, art: 95 };

console.log(Object.keys(scores));     // ["maths", "science", "art"]
console.log(Object.values(scores));   // [88, 72, 95]
console.log(Object.entries(scores));  // [["maths", 88], ["science", 72], ["art", 95]]
```

Each of these returns a real array, unlocking everything from Chapter 6. `entries()` is the most useful for loops — each item is a `[key, value]` pair, which destructures neatly (covered below).

Combining Object.values with reduce

```
const total = Object.values(scores).reduce((sum, score) => sum + score, 0);

console.log(total);   // 255
```

This is the cleaner version of the Fundamentals Chapter 7 inventory challenge, which used a manual `for...in` loop with a separate running-total variable — here, `reduce` does both jobs in one expression.

Array Destructuring

```
const [first, second] = ["Alice", "Bob", "Carl"];

console.log(first);    // "Alice"
console.log(second);   // "Bob"

// Used directly in a loop over entries():
for (const [subject, score] of Object.entries(scores)) {
  console.log(`${subject}: ${score}`);
}
```

Destructuring unpacks array elements straight into named variables in one line. `for...of` (a cousin of the `for...in` from Fundamentals Chapter 4/7) pairs naturally with `entries()`, since each entry is itself a two-element array ready to destructure.

Object Destructuring

```
const person = { name: "Philip", age: 35, city: "London" };

const { name, city } = person;

console.log(name);   // "Philip"
console.log(city);   // "London"
```

Object destructuring pulls named properties straight out into variables of the same name — far shorter than writing `person.name` and `person.city` on separate lines, especially useful for function parameters (see below).

The Spread Operator (...) — Expanding a Collection

```
const nums1 = [1, 2, 3];
const nums2 = [4, 5];

const combined = [...nums1, ...nums2];
console.log(combined);   // [1, 2, 3, 4, 5]

const original = { name: "Philip", age: 35 };
const updated = { ...original, age: 36 };

console.log(updated);   // { name: "Philip", age: 36 } — original is untouched
```

Spread "unpacks" an array or object's contents into a new one. `{ ...original, age: 36 }` copies every property from `original`, then overwrites `age` — a clean way to update one field without mutating the original object, the same non-mutating spirit as `map` / `filter` from Chapter 6.

The Rest Parameter (...) — Collecting the Leftovers

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

console.log(sum(1, 2));          // 3
console.log(sum(1, 2, 3, 4));    // 10
```

The exact same `...` syntax, in a function's parameter list, does the opposite job: it gathers any number of arguments into a single real array (`numbers`), rather than expanding something. This is what lets `sum()` accept any number of arguments instead of a fixed count.

> **SAME SYMBOL, OPPOSITE MEANING — CONTEXT DECIDES WHICH**
>
> `...` in an array/object literal or function *call* is **spread** (expanding). `...` in a function's parameter *definition* is **rest** (collecting). There's no separate keyword — only where it appears tells you which one it is.

TOOL	INPUT	OUTPUT
`Object.keys(obj)`	Object	Array of property names
`Object.values(obj)`	Object	Array of values
`Object.entries(obj)`	Object	Array of [key, value] pairs
`{ ...obj, key: val }`	Object	New object, copied + overridden
`function f(...args)`	Any number of arguments	A single real array, args

Coding Challenges

CHALLENGE 1

Given const stock = { apples: 10, bananas: 5, oranges: 8 }, use Object.entries and a for...of loop with destructuring to log each item as "item: quantity", then use Object.values combined with reduce to log the total quantity.

View solution

CHALLENGE 2

Given const settings = { theme: "dark", fontSize: 14, notifications: true }, use the spread operator to create a new object updatedSettings with fontSize changed to 16, without modifying the original settings object. Log both objects to confirm settings is unchanged.

View solution

CHALLENGE 3

Write a function describeTeam(captain, ...players) that logs the captain's name on its own line, then logs every remaining player using a rest parameter and forEach. Call it with at least 4 names total.

View solution

CHAPTER 1 QUICK REFERENCE (INTERMEDIATE)

- **Object.keys/values/entries** — convert an object into a real array of names/values/pairs
- **Array/object destructuring** — unpack values directly into named variables
- **for...of + Object.entries()** — idiomatic way to loop over an object with both key and value
- **Spread (...)** in a literal/call — expands a collection's contents (copying, merging, overriding)
- **Rest (...)** in a parameter list — collects multiple arguments into one real array
- **Spread never mutates** the original — same non-mutating principle as map/filter
- **Next chapter:** closures — how a function can "remember" variables from where it was created

CHAPTER 2 OF 7

Closures

COURSE 2 · CH 2

Closures: How Functions Remember Their Creation Scope

The mechanism behind the "nested functions see outer variables" behaviour first noticed back in Fundamentals Chapter 5

Fundamentals Chapter 5 showed a nested function reading a variable from its enclosing function, and flagged that this "remembering" behaviour is called a **closure**, promising full coverage later — this is that chapter. A closure is simply a function that keeps access to variables from the scope it was created in, even after that outer scope has technically finished running.

The Basic Closure

```
function makeGreeter(greeting) {
  return function(name) {
    console.log(`${greeting}, ${name}!`);
  };
}

const sayHello = makeGreeter("Hello");
const sayHi = makeGreeter("Hi");

sayHello("Philip");   // "Hello, Philip!"
sayHi("Sam");       // "Hi, Sam!"
```

`makeGreeter` finishes running and returns the inner function — but that inner function still remembers whatever `greeting` was at the moment it was created. `sayHello` and `sayHi` are two completely separate closures, each with its own private copy of `greeting`, even though both were built by the exact same `makeGreeter` function.

This Is Exactly Fundamentals Chapter 6's makeMultiplier Challenge

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

const double = makeMultiplier(2);
console.log(double(5));   // 10 — factor (2) is still remembered
```

That earlier challenge asked *why* the inner function could still access `factor` after `makeMultiplier` had already finished — the answer is exactly this chapter's topic. The returned function "closes over" `factor`, carrying its own private reference to it for as long as the returned function itself exists.

A Closure Holds a Reference, Not a Snapshot

```
function makeCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
}

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

Each call to the returned function mutates the SAME `count` variable — it isn't reset, and it isn't a frozen copy taken at creation time. The closure remembers the actual variable, live, the same way two object references (Fundamentals Chapter 7) point at the same underlying object rather than separate copies.

> **TWO COUNTERS FROM MAKECOUNTER() ARE COMPLETELY INDEPENDENT**
>
> `const counterA = makeCounter(); const counterB = makeCounter();` creates two separate `count` variables — one per call to `makeCounter()` — so calling `counterA()` never affects `counterB()`'s count. Each invocation of the outer function creates a brand-new scope to close over.

The Classic Loop + Closure Pitfall (and Why let Fixes It)

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 100);
}
// Logs: 3, 3, 3 — NOT 0, 1, 2 (because var has no per-iteration scope)

for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 100);
}
// Logs: 0, 1, 2 — let creates a fresh i for each iteration
```

This is the single most famous closure-related bug in JavaScript. With `var` (Fundamentals Chapter 2's "avoid it" keyword), there's only ever one shared `i` across the entire loop — by the time any of the three `setTimeout` callbacks actually runs, the loop has already finished and `i` is `3`. `let` creates a genuinely new `i` binding for each iteration, so each callback's closure captures its own separate value.

Practical Use: Private State with a Module-Style Pattern

```
function createBankAccount(startingBalance) {
  let balance = startingBalance;   // not accessible from outside at all

  return {
    deposit(amount) { balance += amount; },
    withdraw(amount) { balance -= amount; },
    getBalance() { return balance; }
  };
}

const account = createBankAccount(100);
account.deposit(50);
console.log(account.getBalance());   // 150
console.log(account.balance);         // undefined — no direct access exists
```

`balance` is never a property on the returned object — there is no `account.balance` to read or overwrite directly. The only way to affect it is through the three methods that closed over it, giving genuine privacy with nothing extra needed: no special keyword, no class syntax (Intermediate Chapter 3) — just an ordinary closure.

> **THIS IS WHAT CHAPTER 8'S EVENT LISTENERS RELY ON TOO**
>
> Every `addEventListener` callback from Fundamentals Chapter 8 is itself a closure — it routinely reaches back to variables declared outside the handler (a counter, a piece of state) and keeps working correctly across repeated clicks, for exactly the same underlying reason as `makeCounter` above.

Coding Challenges

CHALLENGE 1

Write a function makePowerOf(exponent) that returns a function taking a base number and returning base raised to that exponent (use ** for exponentiation). Create square = makePowerOf(2) and cube = makePowerOf(3), then test both with the same input.

View solution

CHALLENGE 2

Write a function makeIdGenerator() that returns a function with no parameters, returning a new sequential ID each time it's called (starting at 1). Create two SEPARATE generators and call each one a few times to prove their counts don't interfere with each other.

View solution

CHALLENGE 3

Write a function createInventory() returning an object with addItem(name, qty), removeItem(name), and getCount(name) methods, all closing over a private object that the caller can never access directly. Add a few items, remove one, and confirm getCount reflects the changes.

View solution

CHAPTER 2 QUICK REFERENCE

- **Closure** — a function that retains access to variables from its creation scope, even after that scope has "finished"
- **Each call to the outer function** creates a fresh, independent set of variables to close over
- **A closure references the real variable**, not a frozen snapshot — mutations are visible across calls
- **var in a loop + closures** = classic bug; **let** gives each iteration its own binding, fixing it
- **Private state pattern:** variables declared inside an outer function, exposed only via returned methods
- **Event listeners are closures** — they routinely reach back to outer variables across repeated events
- **Next chapter:** ES6 classes — constructors, methods, and inheritance with extends

CHAPTER 3 OF 7

ES6 Classes

COURSE 2 · CH 3

ES6 Classes: Constructors, Methods, and Inheritance with extends

A cleaner syntax for the same object-with-methods pattern from Fundamentals Chapter 7 — plus a real way to share behaviour between related types

Fundamentals Chapter 7 built objects by hand, one at a time, each with its own copy of methods like `greet()` defined inline. `class` syntax (added in the same 2015 update as `let` / `const`) provides a blueprint for creating many similarly-shaped objects, plus a built-in way to extend one type's behaviour from another — JavaScript's equivalent of inheritance.

Defining a Class

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

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

const p = new Person("Philip", 35);
p.greet();   // "Hi, I'm Philip"
```

`constructor(...)` runs automatically whenever `new Person(...)` is called — its job is to set up the new object's starting properties via `this`. Every other method (`greet` here) is defined directly inside the class body, with no `function` keyword and no commas between methods — a more compact form than Fundamentals Chapter 7's object-literal method syntax, but functionally similar underneath.

A CLASS IS REALLY JUST A SPECIAL KIND OF FUNCTION

> Behind the scenes, `class` is mostly a cleaner syntax over JavaScript's existing object/prototype system — `typeof Person` actually returns `"function"`. The new keywords (`class`, `constructor`, `extends`) exist purely for readability; they don't introduce a fundamentally different kind of object.

Multiple Instances, Each With Their Own Data

```
const alice = new Person("Alice", 28);
const bob   = new Person("Bob", 42);

alice.greet();   // "Hi, I'm Alice"
bob.greet();     // "Hi, I'm Bob"
```

Each `new Person(...)` call produces a completely independent object with its own `name`/`age` — but all instances share the exact same `greet` method definition, stored once rather than duplicated per object, which is more memory-efficient than the object-literal approach from Chapter 7 at scale.

Inheritance with extends

```
class Student extends Person {
  constructor(name, age, school) {
    super(name, age);   // calls Person's constructor first
    this.school = school;
  }

  study() {
    console.log(`${this.name} is studying at ${this.school}`);
  }
}

const student = new Student("Sam", 20, "MIT");
student.greet();   // "Hi, I'm Sam" — inherited from Person
student.study();   // "Sam is studying at MIT" — Student's own method
```

`class Student extends Person` makes `Student` a more specific version of `Person` — it automatically gets every method `Person` has (`greet`), plus whatever new methods it adds itself (`study`). `super(name, age)` must be called before `this` can be used at all inside a subclass constructor — it runs the parent class's constructor logic first, so `this.name`/`this.age` get set up correctly before `Student` adds its own `school` property.

> **FORGETTING SUPER() IN A SUBCLASS CONSTRUCTOR IS A HARD ERROR**

If a subclass defines its own `constructor`, it MUST call `super(...)` before referencing `this` anywhere — JavaScript throws a `ReferenceError` immediately otherwise. If a subclass needs no extra setup at all, its `constructor` can simply be omitted entirely; the parent's constructor runs automatically in that case.

Overriding a Method

```
class Student extends Person {
  constructor(name, age, school) {
    super(name, age);
    this.school = school;
  }

  greet() {
    super.greet();   // still runs Person's original greet() first
    console.log(`I study at ${this.school}`);
  }
}

new Student("Sam", 20, "MIT").greet();
// "Hi, I'm Sam"
// "I study at MIT"
```

Defining `greet` again inside `Student` replaces (overrides) the inherited version for any `Student` instance. `super.greet()` inside the override still reaches back to `Person`'s original method — useful when the subclass wants to extend, not completely replace, the parent's behaviour.

Getters — Computed Properties That Look Like Plain Fields

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }

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

const rect = new Rectangle(4, 5);
console.log(rect.area);   // 20 — read like a property, not called like a method
```

`get area()` defines a method that's accessed WITHOUT parentheses — `rect.area`, not `rect.area()` — recalculated fresh every time it's read. Useful for values that are always derivable from existing properties (`width` × `height`) rather than stored separately, avoiding the risk of them drifting out of sync.

CONCEPT	FUNDAMENTALS CH 7 (OBJECT LITERAL)	THIS CHAPTER (CLASS)
Creating an instance	`{ name: ..., greet() {...} }`	`new Person(name)`
Method sharing	Duplicated per object	`Shared once across all instances`
Extending behaviour	No built-in mechanism	`class Sub extends Base`
Computed property	A method called with ()	`get propertyName() – read without ()`

Coding Challenges

CHALLENGE 1

Define a class Animal with a constructor taking name, and a method speak() logging "{name} makes a sound." Create two Animal instances with different names and call speak() on each.

View solution

CHALLENGE 2

Define a class Dog that extends Animal from Challenge 1, overriding speak() to log "{name} barks." instead, while still calling the parent's speak() first using super.speak(). Create a Dog instance and call speak() on it.

View solution

CHALLENGE 3

Define a class Circle with a constructor taking radius, and a getter area returning the circle's area (π × radius^2, using 3.14159 for π) and a getter circumference returning 2 × π × radius. Create an instance and log both computed properties.

View solution

CHAPTER 3 QUICK REFERENCE

- **class Name { constructor(...) {...} method() {...} }** — basic class shape
- **new ClassName(...)** — creates an instance, running the constructor

- **class Sub extends Base** — inherits all of Base's methods automatically
- **super(...)** — calls the parent constructor; required before using this in a subclass constructor
- **super.method()** — calls the parent's version of an overridden method
- **get propertyName() {...}** — a method read without parentheses, like a plain property
- **class is mostly syntax** over JavaScript's existing object/prototype system
- **Next chapter:** this in depth — call/apply/bind, and arrow vs regular function context

CHAPTER 4 OF 7

this In Depth

COURSE 2 · CH 4

this In Depth: call/apply/bind, and Arrow vs Regular Context

Finally answering, precisely, what this refers to in every situation — including the ones that break the simple rule

Fundamentals Chapter 5 and 7 both flagged `this` behaviour without fully resolving it: regular functions get their own `this` determined by how they're called; arrow functions inherit it from their surrounding scope. This chapter makes that precise, covers the situations where `this` goes wrong, and introduces three methods — `call`, `apply`, `bind` — that control it directly.

The Real Rule: this Depends on HOW a Function Is Called

```
const person = {
  name: "Philip",
  greet() { console.log(`Hi, ${this.name}`); }
};

person.greet();          // "Hi, Philip" — called AS A METHOD on person

const greetFn = person.greet;
greetFn();               // "Hi, undefined" — called PLAIN, this is no longer person
```

The exact same function, called two different ways, gets a different `this`. `person.greet()` sets `this` to `person`, because it's called *through* `person`. Once that same function is assigned to `greetFn` and called on its own, there's no object to its left at the call site — `this` falls back to `undefined` (in strict mode/modules) rather than `person`.

THIS IS EXACTLY WHY PASSING A METHOD AS A CALLBACK BREAKS IT

`button.addEventListener("click", person.greet)` (Fundamentals Chapter 8) hands the function over to be called plain later — by the time the browser actually calls it, `this` is no longer `person`. This is the single most common real-world "this" bug.

call() and apply() — Explicitly Setting this for One Call

```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}`);
}

const person = { name: "Philip" };

greet.call(person, "Hello");     // "Hello, Philip" — args listed individually
greet.apply(person, ["Hi"]);     // "Hi, Philip" — args passed as an array
```

`call` and `apply` run a function immediately, with the first argument forced into the role of `this` for that one call — regardless of how the function was originally defined or attached. They're functionally identical except for how the remaining arguments are passed: `call` lists them one by one, `apply` takes them as a single array (similar in spirit to the spread operator from Intermediate Chapter 1).

bind() — Permanently Locking this for Later

```
const person = {
  name: "Philip",
  greet() { console.log(`Hi, ${this.name}`); }
};

const boundGreet = person.greet.bind(person);

boundGreet();   // "Hi, Philip" — this stays person, even called standalone

button.addEventListener("click", boundGreet);   // fixes the Chapter 8 callback bug above
```

Unlike `call` / `apply`, `bind` doesn't run the function immediately — it returns a brand-new function with `this` permanently locked to whatever was passed in, no matter how that new function later gets called. This is the standard fix for the callback problem above: bind the method to its object before handing it off as a callback.

Arrow Functions: No Own this, Inherited Instead

```
const timer = {
  seconds: 0,
  start() {
    setInterval(() => {
      this.seconds++;              // arrow function — "this" is inherited from start()
      console.log(this.seconds);
    }, 1000);
  }
};

timer.start();   // logs 1, 2, 3... — "this" correctly stays "timer" throughout
```

Had the `setInterval` callback been a regular function instead, `this` inside it would be `undefined` (plain-call rule from earlier), breaking `this.seconds++` entirely. The arrow function instead has no `this` of its own — it transparently uses whatever `this` was already in scope at the point it was written, which is `timer`, since `start()` itself was called as `timer.start()`.

> **THE PRACTICAL RULE OF THUMB FROM FUNDAMENTALS CHAPTER 5, NOW FULLY JUSTIFIED**
>
> Use a regular `function` for object methods that need their own `this` (set by however they're called). Use an arrow function for callbacks nested inside a method, specifically so they inherit the surrounding `this` instead of losing it.

Why Arrow Functions Can't Be Fixed with bind/call/apply

```
const arrowGreet = () => console.log(this.name);

arrowGreet.call({ name: "Philip" });   // does NOT work — this is still inherited, ignores call entirely
```

Since an arrow function never has its own `this` to begin with, `call` / `apply` / `bind` have nothing to override — they're silently ignored for the `this` argument specifically. This is the one situation where those three methods simply don't apply.

METHOD	RUNS IMMEDIATELY?	SETS THIS FOR
`fn.call(obj, a, b)`	Yes	That one call, args listed individually
`fn.apply(obj, [a, b])`	Yes	That one call, args as an array
`fn.bind(obj)`	No — returns a new function	Every future call of the returned function

Coding Challenges

CHALLENGE 1

Write a plain function introduce(role) that logs `${this.name} works as a ${role}`. Create two different objects, each with a name property, and use call() to invoke introduce with each object as this, passing a different role string each time.

View solution

CHALLENGE 2

Create an object counter with a count property (0) and a method increment() that increases count by 1 and logs it. Extract increment as a standalone function, use bind() to lock it to counter, then call the bound version three times via setTimeout to prove this stays correct even when called asynchronously.

View solution

CHALLENGE 3

Create an object stopwatch with elapsed: 0 and a method start() that uses setInterval with an ARROW function to increment elapsed and log it every second, for 3 seconds (then stop with clearInterval). Confirm elapsed actually increases by logging it.

View solution

CHAPTER 4 QUICK REFERENCE

- **this depends on the call site** — obj.method() sets this to obj; a plain call doesn't
- **fn.call(obj, a, b)** — runs fn immediately with this = obj, args listed individually
- **fn.apply(obj, [a, b])** — same as call, but args passed as an array
- **fn.bind(obj)** — returns a NEW function with this permanently locked to obj
- **Regular functions** get their own this; **arrow functions** inherit this from their surrounding scope
- **call/apply/bind have no effect** on an arrow function's this — there's nothing to override
- **Passing a method as a bare callback** loses its this — bind it first, or use an arrow wrapper
- **Next chapter:** error handling — try/catch/finally and custom Error classes

CHAPTER 5 OF 7

Error Handling

COURSE 2 · CH 5

Error Handling: try/catch/finally and Custom Error Classes

Fundamentals Chapter 10 used try/catch around fetch — this chapter covers the full mechanism, including how to throw and design your own errors

Fundamentals Chapter 10 wrapped `await fetch(...)` in `try/catch` without explaining the mechanism fully. This chapter covers `throw`, the complete `try/catch/finally` structure, and — using Chapter 3's `class` / `extends` syntax — how to build custom error types carrying more information than a plain message string.

throw — Raising an Error Deliberately

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero");
  }
  return a / b;
}

divide(10, 0);   // Uncaught Error: Cannot divide by zero — stops execution entirely
```

`throw new Error("message")` immediately stops the current function (and everything that called it) from continuing normally — unlike Go's `error` return values (a deliberate contrast worth knowing if the Go course is ever revisited), JavaScript errors propagate upward automatically until something explicitly catches them.

try/catch — Catching a Thrown Error

```
try {
  const result = divide(10, 0);
  console.log(result);   // never runs — divide already threw
} catch (error) {
  console.log("Something went wrong:", error.message);
}
// "Something went wrong: Cannot divide by zero"
```

Code inside `try { }` runs normally until something throws — at that exact point, execution jumps straight into `catch (error) { }`, skipping anything remaining in the `try` block. `error.message` holds the text passed to `new Error(...)`; every error also has a `name` property (`"Error"` by default).

finally — Runs No Matter What

```
function loadData() {
  console.log("Loading started");
  try {
    throw new Error("Network failure");
  } catch (error) {
    console.log("Caught:", error.message);
  } finally {
    console.log("Loading finished");   // runs whether or not an error was thrown
  }
}
```

`finally { }` runs after `try` / `catch` complete, regardless of whether an error occurred — useful for cleanup work (hiding a loading spinner, closing a connection) that needs to happen either way. This is conceptually similar to Go's `defer` from the Go Intermediate course, just scoped specifically to error handling rather than every function exit.

Custom Error Classes

```
class ValidationError extends Error {
  constructor(message, field) {
    super(message);          // sets up the standard message/stack behaviour
    this.name = "ValidationError";
    this.field = field;
  }
}

function validateAge(age) {
  if (age < 0) {
    throw new ValidationError("Age cannot be negative", "age");
  }
}
```

`class ValidationError extends Error` (Chapter 3's inheritance, applied to the built-in `Error` type) creates a genuinely new error type carrying extra structured data — here, `field` — alongside the standard `message`. `super(message)` must run first, exactly as with any other subclass constructor, to set up `Error`'s own internal behaviour (including the stack trace).

Distinguishing Error Types in a catch Block

```
try {
  validateAge(-5);
} catch (error) {
  if (error instanceof ValidationError) {
    console.log(`Invalid field "${error.field}": ${error.message}`);
  } else {
    console.log("Unexpected error:", error.message);
  }
}
// Invalid field "age": Age cannot be negative
```

`error instanceof ValidationError` checks whether the caught error is specifically that custom type (or a subclass of it) — this is JavaScript's rough equivalent of Go's `errors.As` from the Go Advanced course, letting a `catch` block react differently depending on exactly what went wrong, rather than treating every error identically.

A CATCH BLOCK WITH NO PARAMETER CATCHES EVERYTHING INDISCRIMINATELY

Without checking `instanceof`, a single `catch` block treats a genuine validation problem the same as an unrelated bug (a typo causing a `TypeError`, for instance) — both get silently swallowed and handled identically. Checking the specific error type before deciding how to respond avoids masking real bugs as if they were expected validation failures.

PIECE	PURPOSE
`throw new Error("msg")`	Deliberately raise an error, halting normal execution
`try { } catch (e) { }`	Run code; if it throws, jump to catch instead of crashing
`finally { }`	Always runs after try/catch, error or not — for cleanup
`class X extends Error`	Define a custom error type with extra data fields
`error instanceof X`	Check which specific error type was actually caught

Coding Challenges

CHALLENGE 1

Write a function parsePositiveNumber(value) that throws a plain Error if value is negative or not a number (use isNaN), otherwise returns it. Call it inside a try/catch with a value that triggers the error, logging the caught message.

View solution

CHALLENGE 2

Define a custom error class InsufficientFundsError extending Error, with a constructor taking (message, shortfall) and storing shortfall. Write a function withdraw(balance, amount) that throws it if amount > balance, including the shortfall (amount - balance). Catch it and log a message using error.shortfall.

View solution

CHALLENGE 3

Write a function riskyOperation(shouldFail) that throws a RangeError if shouldFail is true, otherwise returns "Success". Wrap a call in try/catch/finally: log the result or the error's name+message in catch, and always log "Operation complete" in finally, regardless of outcome. Call it twice, once with each boolean.

View solution

CHAPTER 5 QUICK REFERENCE

- **throw new Error("message")** — deliberately raises an error, halting normal flow
- **try { } catch (error) { }** — catches a thrown error instead of letting it crash the program
- **finally { }** — always runs after try/catch, error or not

- **error.message** — the text passed when the error was created; **error.name** — its type name
- **class X extends Error** — custom error type; must call super(message) first
- **error instanceof X** — checks the specific error type caught, for differentiated handling
- **Built-in error types:** Error, TypeError, RangeError, SyntaxError, and more
- **Next chapter:** array/object copying — shallow vs deep, and common mutation pitfalls

CHAPTER 6 OF 7

Array/Object Copying

COURSE 2 · CH 6

Array/Object Copying: Shallow vs Deep, and Mutation Pitfalls

Why spreading an object doesn't always actually copy it — the gap between "looks copied" and "is independent"

Intermediate Chapter 1 introduced the spread operator for copying arrays and objects, describing it as non-mutating, "the same spirit as map/filter." This chapter shows exactly where that breaks down: spread only copies one level deep, and anything nested still points at the original.

Reference Types: The Root of the Problem

```
const a = [1, 2, 3];
const b = a;   // NOT a copy — b points at the SAME array as a

b.push(4);
console.log(a);   // [1, 2, 3, 4] — a changed too!
```

Arrays and objects are **reference types** — a variable holding one doesn't hold the actual data, it holds a reference (an address) to where that data lives. `const b = a` copies the reference, not the array — both variables end up pointing at the exact same underlying array, so mutating through either one is visible through both.

Shallow Copying with Spread

```
const original = { name: "Philip", age: 35 };
const copy = { ...original };

copy.age = 36;
console.log(original.age);   // 35 — unaffected, this top level is a real copy
```

`{ ...original }` genuinely does create a separate object — for top-level, primitive values (strings, numbers, booleans) this is a complete, independent copy. This is "shallow" because it only copies one level deep; the

moment a property's value is itself an object or array, the problem returns.

Where Shallow Copying Breaks Down

```
const original = {
  name: "Philip",
  address: { city: "London" }
};

const copy = { ...original };
copy.address.city = "Paris";

console.log(original.address.city);   // "Paris" — the ORIGINAL changed too!
```

Spread copied the `address` *property*, but that property's value is itself an object — spread copies the reference to that nested object, not the nested object's contents. `copy.address` and `original.address` still point at the exact same inner object, so mutating one mutates both. This is the most common real-world "spread didn't work" bug.

> **THE SAME PROBLEM APPLIES TO ARRAYS OF OBJECTS**
> `const copy = [...original]` creates a new array, but if `original` contains objects, each element in `copy` still points at the SAME objects as `original` — changing a property on `copy[0]` changes `original[0]` too. Spreading an array is shallow in exactly the same way as spreading an object.

Deep Copying with structuredClone

```
const original = {
  name: "Philip",
  address: { city: "London" }
};

const copy = structuredClone(original);
copy.address.city = "Paris";

console.log(original.address.city);   // "London" — genuinely independent now
```

`structuredClone()` is a built-in browser/Node function that performs a true **deep copy** — it recursively copies every nested object and array, so the result is completely independent of the original at every level, no matter how deeply nested.

> **STRUCTUREDCLONE HAS LIMITS**

It can't clone functions or DOM elements (Fundamentals Chapter 8) — attempting to clone an object containing either throws an error. For ordinary data (objects, arrays, strings, numbers, dates, Maps, Sets), it's the standard modern tool, replacing older workarounds like `JSON.parse(JSON.stringify(obj))`, which silently loses things like `undefined` values and dates.

map() Avoids This Problem Entirely

```
const people = [
  { name: "Alice", age: 28 },
  { name: "Bob", age: 42 }
];

const updated = people.map(person => ({ ...person, age: person.age + 1 }));

console.log(people[0].age);    // 28 — original untouched
console.log(updated[0].age);   // 29
```

Combining `map` (Fundamentals Chapter 6) with object spread inside the callback creates a genuinely new object for each element — since `name` and `age` are both primitives here, this shallow approach is entirely sufficient; deep copying is only needed when the nested values are themselves objects/arrays that also need protecting from mutation.

OPERATION	TOP-LEVEL COPY?	NESTED OBJECTS/ARRAYS COPIED?
`const b = a`	No — same reference	No
`{ ...obj } / [...arr]`	Yes (shallow)	No — nested values still shared
`structuredClone(obj)`	Yes	Yes — fully independent at every level

Coding Challenges

CHALLENGE 1

Create const original = [1, 2, 3]. Create const sameRef = original (no copy) and const shallowCopy = [...original]. Push 4 onto sameRef and 5 onto shallowCopy, then log original, sameRef, and shallowCopy to show the different effects.

View solution

CHALLENGE 2

Create const settings = { theme: "dark", display: { brightness: 80 } }. Create a shallow copy with spread, change the copy's display.brightness to 50, then log the original's display.brightness to demonstrate the shallow-copy bug from

this chapter.

View solution

CHALLENGE 3

Repeat Challenge 2, but use structuredClone instead of spread to create the copy. Change the copy's display.brightness to 50, then log the original's display.brightness to confirm it's now unaffected.

View solution

CHAPTER 6 QUICK REFERENCE

- **const b = a** (arrays/objects) — copies the reference only; both point at the same data
- **{ ...obj } / [...arr]** — shallow copy; top level is independent, nested values are NOT
- **structuredClone(obj)** — true deep copy; every level is fully independent
- **structuredClone cannot clone functions or DOM elements** — ordinary data only
- **map() + spread per item** — a common, sufficient pattern when elements only hold primitives
- **Mutating a nested shared reference** is the most common real-world "spread didn't work" bug
- **Next chapter:** local storage and JSON — persisting data in the browser between page loads

CHAPTER 7 OF 7

Local Storage and JSON

COURSE 2 · CH 7

Local Storage and JSON: Persisting Data Between Page Loads

Everything in memory disappears on refresh — localStorage is the simplest way to make data survive

Every variable in every chapter so far has vanished the instant the page reloads. `localStorage` is a browser-provided key/value store that survives refreshes, browser restarts, even the computer rebooting — and combined with `JSON.stringify` / `JSON.parse` (the same JSON mechanics behind Fundamentals Chapter 10's `fetch`), it can store more than just plain text.

Storing and Reading a String

```
localStorage.setItem("username", "Philip");

const name = localStorage.getItem("username");
console.log(name);   // "Philip" — still there even after a page refresh
```

`setItem(key, value)` saves a value under a string key; `getItem(key)` reads it back. Reload the page and run `getItem("username")` again — the value is still there, unlike every variable from earlier chapters, which would be reset to nothing on reload.

LOCALSTORAGE ONLY STORES STRINGS

`localStorage.setItem("age", 35)` silently converts `35` to the string `"35"` — reading it back with `getItem` gives a string, not a number. Trying to store an object directly (`setItem("user", { name: "Philip" })`) doesn't work either: it gets converted to the unhelpful string `"[object Object]"`, losing all the actual data.

JSON.stringify — Turning Data Into a Storable String

```
const user = { name: "Philip", age: 35, isStudent: true };

const jsonString = JSON.stringify(user);
console.log(jsonString);   // '{"name":"Philip","age":35,"isStudent":true}'

localStorage.setItem("user", jsonString);
```

`JSON.stringify` converts an object or array into a JSON-formatted string — text that fully represents the original data's structure and values, safe to store as a single `localStorage` string. This is the same conversion that happens automatically inside a `fetch` request body when sending JSON data to a server.

JSON.parse — Turning It Back Into Real Data

```
const jsonString = localStorage.getItem("user");
const user = JSON.parse(jsonString);

console.log(user.name);   // "Philip" — a real object again, with real dot-notation access
console.log(typeof user.age);   // "number" — NOT a string, unlike storing a number directly
```

`JSON.parse` does the reverse: a JSON string back into a real, usable JavaScript object — with correct types preserved (`age` comes back as an actual number, not the string `"35"`). This `stringify` / `parse` pair is exactly how Fundamentals Chapter 10's `response.json()` works internally.

The Complete Save/Load Pattern

```
function saveSettings(settings) {
  localStorage.setItem("settings", JSON.stringify(settings));
}

function loadSettings() {
  const stored = localStorage.getItem("settings");
  return stored ? JSON.parse(stored) : null;   // null if nothing was ever saved
}

saveSettings({ theme: "dark", fontSize: 16 });

const settings = loadSettings();
console.log(settings.theme);   // "dark"
```

`localStorage.getItem` returns `null` if the key was never set — checking for that with `stored ? ... : null` (Fundamentals Chapter 3's ternary) avoids calling `JSON.parse(null)`, which would throw rather than returning something sensible.

Removing and Clearing Storage

```
localStorage.removeItem("username");   // removes just one key
localStorage.clear();                  // removes EVERYTHING this site has stored
```

`removeItem` deletes a single key, the direct equivalent of `delete obj.key` from Fundamentals Chapter 7; `clear()` wipes every key the current site has stored in `localStorage` entirely.

LOCALSTORAGE VS SESSIONSTORAGE

`localStorage` persists indefinitely, across browser restarts, until explicitly cleared. `sessionStorage` shares the exact same API (`setItem` / `getItem` / `removeItem` / `clear`) but is wiped automatically when the browser tab closes — useful for data that should only last for the current visit.

TOOL	PURPOSE
`localStorage.setItem(key, value)`	Save a string value, persisting across reloads
`localStorage.getItem(key)`	Read a stored value; returns null if never set
`JSON.stringify(value)`	Convert an object/array into a storable JSON string
`JSON.parse(jsonString)`	Convert a JSON string back into a real object/array
`localStorage.removeItem(key) / .clear()`	Delete one key / delete everything

Coding Challenges

CHALLENGE 1

Save the string "dark" under the key "theme" using localStorage.setItem. Read it back with getItem and log it. Then use removeItem to delete it, and log getItem("theme") again to confirm it now returns null.

View solution

CHALLENGE 2

Create an object task = { title: "Buy milk", done: false }. Save it to localStorage under "task1" using JSON.stringify. Read it back, parse it with JSON.parse, and log task.title and typeof task.done to confirm the boolean type survived the round trip.

View solution

CHALLENGE 3

Write functions saveTodos(todos) and loadTodos() using the save/load pattern from this chapter, where todos is an array of strings. saveTodos should stringify and store the array; loadTodos should return the parsed array, or an empty array [] (not null) if nothing has been saved yet. Save 3 todos, then load and log them.

View solution

CHAPTER 7 QUICK REFERENCE

- **localStorage.setItem(key, value)** — saves a STRING, persisting across reloads/restarts
- **localStorage.getItem(key)** — reads it back; returns null if the key was never set
- **JSON.stringify(value)** — converts an object/array into a JSON string for storage
- **JSON.parse(jsonString)** — converts a JSON string back into a real object/array, types intact
- **localStorage.removeItem(key)** — deletes one key; **clear()** — deletes everything for this site
- **sessionStorage** — same API, but wiped when the browser tab closes
- **Always guard JSON.parse** against a null/missing value before calling it
- **This completes JavaScript Intermediate.** Advanced begins with modules — import/export and bundlers conceptually.