JavaScript Fundamentals

A Complete 10-Chapter Course

Topics covered:

Basics & Console · Variables & Types · Operators & Control Flow · Loops

Functions · Arrays · Objects · The DOM · Forms & Input · Async JS

Exercises: 30 hands-on challenges with sample solutions

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

Table of Contents

CHAPTER 1 OF 10

What JS Is, Where It Runs, Embedding in HTML, the Console

COURSE 1 · CH 1

What JS Is, Where It Runs, and the Console

The one language every browser understands natively — and the tool you'll use constantly while learning it

JavaScript is the programming language that runs inside web browsers, making pages interactive — reacting to clicks, updating content without reloading, validating a form before it's submitted. Unlike PHP (covered in the previous series), which runs on the **server** before a page is sent to the visitor, JavaScript typically runs directly in the visitor's own browser, on their machine.

Where JavaScript Runs

Browser (client-side)
The original, most common use — runs on the visitor's device

Node.js (server-side)
JS running on a server instead — covered in Advanced Chapter 6

Browser DevTools console
Where you'll experiment constantly throughout this course

This course focuses entirely on browser JavaScript first — exactly where most beginners start, and where the immediate, visible feedback of seeing a page react makes learning genuinely engaging.

Embedding JavaScript in HTML — Three Ways

1. Inline (avoid this, shown for completeness)

```
<button onclick="alert('Hello!')">Click me</button>
```

2. Internal — inside a <script> tag

```
<script>
  console.log("Hello from internal JS!");
</script>
```

3. External — a separate .js file (the standard approach)

```
<!-- in your HTML file -->
<script src="script.js"></script>

// in script.js
console.log("Hello from an external file!");
```

External files are the standard approach in real projects — they keep HTML and JavaScript separated (mirroring how CSS is usually kept in its own file too), and the same script file can be reused across multiple HTML pages.

WHERE THE <SCRIPT> TAG GOES MATTERS

A script placed in the `<head>` runs before the page's HTML content has loaded — trying to interact with an element that doesn't exist yet (Chapter 8's DOM manipulation) will fail. Placing `<script>` tags just before the closing `</body>` tag, or using the `defer` attribute, ensures the page's content exists first.

console.log() — Your Most-Used Tool

```
console.log("Hello, world!");
console.log(42);
console.log(2 + 3);
```

`console.log()` prints a value to the browser's developer console — opened with F12 or right-click → Inspect → Console tab. It's the JavaScript equivalent of PHP's `echo` / `var_dump()` from the previous course series, and you'll use it constantly while learning and debugging.

```
> console.log("Hello, world!");
Hello, world!

> 2 + 3
5

> typeof "hello"
'string'
```

The console can also be typed into directly, like a small interactive scratchpad — typing an expression and pressing Enter immediately shows its result, without needing a full HTML page or script file at all.

OTHER USEFUL CONSOLE METHODS

`console.error("message")` highlights output in red, useful for genuine errors. `console.warn("message")` shows a yellow warning. `console.table(someArray)` displays array/object data in a readable table — genuinely handy once Chapter 6's arrays are introduced.

Statements and Semicolons

```
console.log("First statement");
console.log("Second statement");
```

Each instruction is a "statement," conventionally ended with a semicolon — similar to PHP. JavaScript can technically infer missing semicolons in many cases ("automatic semicolon insertion"), but this has subtle edge cases that can cause confusing bugs; this course writes semicolons explicitly throughout, which is the safer, more common professional habit.

Comments

```
// a single-line comment

/* a multi-line
   comment block */

console.log("This runs"); // comments can follow code too
```

Coding Challenges

CHALLENGE 1

Create an HTML file with an external script.js file linked just before the closing </body> tag. Inside script.js, use console.log() to print your name, then a simple calculation like 7 * 6, on two separate lines. Open the page and check the result in the browser console.

View solution

CHALLENGE 2

In the browser console directly (no file needed), try console.log(), console.warn(), and console.error() each with a different short message, and note in a comment what visually distinguishes each one in the console's output.

View solution

CHALLENGE 3

Write a script.js file with three console.log() statements, each preceded by a comment explaining what the line below it does (practice writing both single-line comment styles). Include at least one multi-line /* */ comment describing the

whole file's purpose at the top.

View solution

CHAPTER 1 QUICK REFERENCE

- **JavaScript runs in the browser** by default (client-side) — different from PHP, which runs on the server
- **<script src="file.js"></script>** — the standard way to include JS; place near the end of <body>, or use defer
- **console.log()** — prints a value to the browser's DevTools console; your most-used debugging tool
- **console.error() / console.warn()** — visually distinct variants for errors/warnings
- **Statements end with a semicolon** — write them explicitly, despite automatic insertion existing
- **// and /* */** — single-line and multi-line comments
- **Next chapter:** variables and data types — var, let, const

CHAPTER 2 OF 10

Variables, Data Types, var/let/const

COURSE 1 · CH 2

Variables, Data Types, and var/let/const

Storing values with a name, and the three different keywords JavaScript gives you to do it

Chapter 1 used `console.log("Hello, JS!")` directly on a literal value. Real code almost always needs to store a value first, under a name, so it can be reused and changed — that's what a **variable** is. JavaScript has three keywords for declaring one: `var`, `let`, and `const` — only two of which are worth reaching for in modern code.

Declaring a Variable

```
let name = "Philip";
let age = 35;

console.log(name);   // "Philip"
console.log(age);    // 35
```

`let name = "Philip";` creates a variable called `name`, holding the value `"Philip"`. There's no need to declare what *type* of value it will hold up front — JavaScript figures that out automatically from whatever is assigned.

let vs const — Can It Be Reassigned?

```
let score = 0;
score = 10;       // fine — let allows reassignment

const pi = 3.14159;
pi = 4;           // TypeError: Assignment to constant variable
```

`let` can be reassigned later; `const` cannot — trying to assign a new value to a `const` variable throws an error immediately. The rule of thumb: default to `const` unless the value genuinely needs to change later, then switch to `let`. This makes code easier to reason about, since a `const` guarantees that value never changes anywhere below its declaration.

VAR STILL EXISTS, BUT AVOID IT

`var` is the original way to declare a variable, predating `let` / `const` (added in 2015). It has looser, more error-prone scoping rules — a `var` declared inside an `if` block leaks out into the surrounding function, unlike `let` / `const`, which stay neatly contained to the block they're declared in. Modern code has no good reason to use `var`.

JavaScript's Core Data Types

```
let count = 42;            // number — both whole numbers and decimals
let price = 19.99;         // number — same type, no separate "float"
let label = "Hello";       // string — text, single or double quotes
let active = true;         // boolean — true or false
let nothing = null;        // null — deliberately "no value"
let notSet;                // undefined — declared, but never given a value
```

Unlike languages with separate integer/decimal types, JavaScript has just one `number` type covering both. `null` and `undefined` are subtly different: `undefined` means a variable was never assigned anything; `null` means a value was deliberately set to "nothing" on purpose.

Checking a Value's Type

```
console.log(typeof 42);          // "number"
console.log(typeof "hello");     // "string"
console.log(typeof true);        // "boolean"
console.log(typeof undefined);   // "undefined"
```

`typeof` returns a string naming a value's type — useful while learning, and occasionally in real code when a value's type genuinely isn't known in advance.

Template Literals — Building Strings with Variables

```
let name = "Philip";
let age = 35;

console.log(`Hello, my name is ${name} and I'm ${age}.`);
// Hello, my name is Philip and I'm 35.
```

Backticks (`` ` ``) instead of quotes create a **template literal**, allowing `${ }` to embed variables (or any expression) directly inside a string — far more readable than joining strings together with `+`.

KEYWORD	REASSIGNABLE?	USE IT WHEN...
`const`	No	The default choice — value won't change
`let`	Yes	The value genuinely needs to change later
`var`	Yes	Avoid — kept only for reading old code

Coding Challenges

CHALLENGE 1

Declare a const city with your city's name, and a let temperature with a number. Log both using a single template literal in the form "It is X degrees in Y".

View solution

CHALLENGE 2

Declare let score = 0. Reassign it three times (e.g. += 10 each time), logging score after each reassignment. Then declare const maxScore = 100 and try reassigning it, observing the error in the console.

View solution

CHALLENGE 3

Declare one variable of each core type (number, string, boolean, null, undefined) and use typeof to log each variable's type alongside its value.

View solution

CHAPTER 2 QUICK REFERENCE

- **const** — cannot be reassigned; the default choice
- **let** — can be reassigned; use only when the value will actually change
- **var** — older, loosely-scoped; avoid in modern code
- **Core types:** number, string, boolean, null, undefined
- **typeof value** — returns a string naming that value's type

- **null** — deliberately "no value"; **undefined** — never assigned a value
- **Template literals:** `text ${variable} more text` — backticks, not quotes
- **Next chapter:** operators and control flow — if/else, switch, ternary

CHAPTER 3 OF 10

Operators and Control Flow

COURSE 1 · CH 3

Operators and Control Flow: if/else, switch, Ternary

Comparing values and making decisions — including JavaScript's own take on loose vs strict equality

Decisions in JavaScript will look immediately familiar coming from PHP — the syntax is nearly identical. The genuinely important new concept here is **truthy and falsy** values, and the strong recommendation to use `===` rather than `==` almost universally.

Comparison Operators

OPERATOR	MEANING
`===`	Strict equality — value AND type must match, no conversion
`!==`	Strict inequality
`==`	Loose equality — converts types before comparing (avoid)
`!=`	Loose inequality (avoid)
`> / < / >= / <=`	Greater/less than, and "or equal" variants

```
console.log(5 === "5");   // false — different types, no conversion
console.log(5 == "5");    // true — "5" is converted to a number first
console.log(0 == false);  // true — surprising, and exactly why == is discouraged
```

USE === ALMOST ALWAYS — THIS IS EVEN MORE STRONGLY RECOMMENDED IN JS THAN IN PHP

JavaScript's `==` performs type coercion with rules that produce some genuinely surprising results (`0 == false` , `"" == 0` , `null == undefined`). This is the JavaScript equivalent of PHP Fundamentals Chapter 2's `==` vs `===` guidance, but the JS community treats it as even more of a hard rule — production JS code overwhelmingly uses `===` / `!==` by default, reaching for `==` only in rare, deliberate cases.

Truthy and Falsy — Values Treated as true/false in a Condition

Falsy (treated as false)

```
false
```

```
0
```

```
""
```

(empty string)

```
null
```

```
undefined
```

```
NaN
```

Truthy (everything else)

```
true
```

```
1, -1, 3.14
```

(any non-zero number)

```
"hello", "0", " "
```

(any non-empty string!)

```
[]
```

(an empty array — still truthy!)

```
{}
```

(an empty object — still truthy!)

```
if ("0") {
  console.log("This runs!");   // "0" is a non-empty STRING — truthy, even though it looks like zero
}
if ([]) {
  console.log("This also runs!");  // an empty array is still truthy
}
```

THE STRING "0" AND AN EMPTY ARRAY [] ARE BOTH GENUINELY TRUTHY — A VERY COMMON SURPRISE

There is no special-casing for "looks like zero" or "looks empty" — only the exact six falsy values listed above are ever falsy. Every other value, including `"0"` and `[]`, is truthy, full stop.

if / else if / else

```
const score = 75;

if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 70) {
  console.log("Grade: B");
} else {
  console.log("Grade: F");
}
```

Note `else if` is always two words in JavaScript — unlike PHP, there's no single-word `elseif` equivalent here.

The Ternary Operator

```
const age = 20;
const status = age >= 18 ? "adult" : "minor";
console.log(status);  // "adult"
```

Identical syntax and use case to PHP's ternary from Fundamentals Chapter 3 — a compact if/else for simple value assignments.

The Logical OR/AND "Default Value" Trick

```
function greet(name) {
  const displayName = name || "Guest";   // if name is falsy, use "Guest" instead
  console.log(`Hello, ${displayName}!`);
}

greet("Philip");  // "Hello, Philip!"
greet();          // "Hello, Guest!" — name is undefined (falsy), so the fallback is used
```

?? IS A SAFER ALTERNATIVE — ONLY FALLS BACK FOR NULL/UNDEFINED, NOT EVERY FALSY VALUE

`const displayName = name ?? "Guest";` — the "nullish coalescing" operator only uses the fallback when the left side is specifically `null` or `undefined`, not for other falsy values like `0` or `""`. This avoids the surprising case where a genuinely valid value like `0` or an empty string gets incorrectly replaced by `||`'s fallback. Covered fully in Intermediate Chapter 1.

switch

```
const day = "Wednesday";

switch (day) {
  case "Monday":
    console.log("Start of the week");
    break;
  case "Wednesday":
    console.log("Midweek");
    break;
  default:
    console.log("Just another day");
}
```

Behaves identically to PHP's switch (Fundamentals Chapter 3), including the same fall-through behaviour when `break` is omitted, and JavaScript's `switch` always compares using strict equality (`===`) internally — never the loose comparison rules from `==`.

Coding Challenges

CHALLENGE 1

Write five console.log() statements testing whether each of these values is truthy or falsy inside an if/else: 0, "", "false" (the string), null, and [] (empty array). Predict each result first, then verify.

View solution

CHALLENGE 2

Write a function describeTemperature(celsius) using if/else if/else that returns "Freezing", "Cold", "Mild", or "Hot" based on reasonable thresholds. Test it with four different values and console.log() each result.

View solution

CHALLENGE 3

Write a function getDiscount(customerType) using switch that returns 0.2 for "vip", 0.1 for "member", and 0 for anything else (default). Then write a function welcomeMessage(name) using the || fallback trick to default to "Guest" if name is falsy, and test both functions with a few different inputs.

View solution

CHAPTER 3 QUICK REFERENCE

- **=== / !==** — strict comparison, use almost always; **== / !=** — loose, avoid
- **Falsy values (only six):** false, 0, "", null, undefined, NaN — everything else is truthy
- **"0" and [] are truthy** — a very common surprise; only the exact six falsy values count
- **if / else if / else** — else if is always two words in JS
- **Ternary:** condition ? a : b
- **value || fallback** — uses fallback for ANY falsy value; **value ?? fallback** — only for null/undefined (Intermediate Ch 1)
- **switch** — same fall-through behaviour as PHP; always compares strictly internally
- **Next chapter:** loops — for, while, for...of, for...in

CHAPTER 4 OF 10

Loops

COURSE 1 · CH 4

Loops: for, while, for...of, for...in

Four loop styles — two should feel familiar from PHP, two are genuinely new and JavaScript-specific

`for` and `while` work essentially identically to PHP. `for...of` and `for...in` are new — and easily confused with each other, since their names look so similar despite doing genuinely different things.

for — Known Repeat Count

```
for (let i = 1; i <= 5; i++) {
  console.log(`Count: ${i}`);
}
```

Identical structure to PHP Fundamentals Chapter 4 — initialiser, condition, step. Note `let i` rather than a plain variable — declaring the loop counter with `let` keeps it properly block-scoped to the loop.

while / do...while

```
let attempts = 0;

while (attempts < 3) {
  attempts++;
  console.log(`Attempt ${attempts}`);
}
```

Same behaviour as PHP — checks the condition before each iteration. `do...while` also exists in JavaScript with identical "runs at least once" semantics.

for...of — Iterating Over Values (Arrays, Strings)

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

for (const colour of colours) {
  console.log(colour);
}
// "red"
// "green"
// "blue"
```

`for...of` gives you each **value** directly, one at a time — the closest JavaScript equivalent to PHP's `foreach ($array as $value)` from Fundamentals Chapter 4. It works on arrays, strings (iterating character by character), and several other "iterable" types covered later in the series.

for...in — Iterating Over Keys/Property Names

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

for (const key in person) {
  console.log(`${key}: ${person[key]}`);
}
// "name: Philip"
// "age: 35"
```

`for...in` gives you each **key** (property name) — designed for plain objects, where there's no single "value" to iterate directly the way an array has. Objects are covered fully in Chapter 7; this is enough to use `for...in` productively now.

> **DON'T USE FOR...IN ON ARRAYS — USE FOR...OF INSTEAD**
>
> `for...in` on an array technically works, but iterates over the array's *indexes* (as strings: "0", "1", "2"), not its values — and can also pick up unexpected extra properties in some situations. The clear rule: arrays → `for...of` (or array methods, Chapter 6). Plain objects → `for...in` (or `Object.keys()`, also Chapter 7).

break and continue

```
for (let i = 1; i <= 10; i++) {
  if (i === 7) {
    break;       // exits the loop entirely
  }
  if (i % 2 === 0) {
    continue;    // skips to the next iteration
  }
  console.log(i);
}
// 1, 3, 5
```

Behave identically to PHP — `break` exits the loop, `continue` skips to the next iteration.

LOOP	BEST FOR
for	Known/calculable repeat count
while / do...while	Unknown count, condition-driven
for...of	Iterating array VALUES (or string characters)
for...in	Iterating object KEYS/property names

A PREVIEW: ARRAY METHODS OFTEN REPLACE LOOPS ENTIRELY

Chapter 6 introduces `map()`, `filter()`, and `forEach()` — methods that handle many common "loop over an array and do something" tasks more concisely than a manual loop. Both approaches matter: loops are more flexible and universal, array methods are often clearer for simple transformations.

Coding Challenges

CHALLENGE 1

Use a for loop to console.log() the 9 times table from 9×1 to 9×12, one line per result.

View solution

CHALLENGE 2

Given const fruits = ["apple", "banana", "cherry", "date"], use for...of to log each fruit, and a separate for...in loop on const car = { make: "Toyota", model: "Corolla", year: 2022 } to log each key and its value. Add a comment explaining why for...in would be the wrong choice for the fruits array.

View solution

CHALLENGE 3

Use a while loop to find and log the first power of 2 that exceeds 1000 (i.e. keep doubling a starting value of 1 until it's greater than 1000), logging each doubled value along the way.

View solution

CHAPTER 4 QUICK REFERENCE

- **for (init; condition; step)** — known/calculable repeat count, same as PHP
- **while / do...while** — condition-driven, same as PHP
- **for...of** — iterates VALUES of an array/string; closest equivalent to PHP's foreach
- **for...in** — iterates KEYS of a plain object; do NOT use on arrays
- **break / continue** — same behaviour as PHP
- **Array methods (map/filter/forEach, Ch 6)** often replace manual loops for simple cases
- **Next chapter:** functions — declarations, expressions, arrow functions, scope

CHAPTER 5 OF 10

Functions

COURSE 1 · CH 5

Functions: Declarations, Expressions, Arrow Functions, Scope

Three different ways to write a function in JavaScript — and why the differences genuinely matter, not just stylistically

PHP Fundamentals Chapter 5 covered one way to define a function. JavaScript has three distinct syntaxes, each with real behavioural differences — not just three ways to write the same thing. This chapter covers all three, plus scope, which works similarly to PHP but with one important new wrinkle.

Function Declarations

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

greet("Philip");   // "Hello, Philip!"
```

The most familiar style — looks almost identical to PHP's function syntax (Fundamentals Chapter 5), minus the `$` prefix on parameters.

Function Expressions

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

greet("Philip");
```

Here the function itself is treated as a **value**, assigned to a `const` variable — exactly like assigning a number or string. This reflects something genuinely fundamental about JavaScript: functions are values, storable in variables, passable as arguments, returnable from other functions ("first-class functions"). Chapter 6's array methods (`map`, `filter`) rely entirely on this.

Arrow Functions

```
const greet = (name) => {
  console.log(`Hello, ${name}!`);
};

// Single parameter: parentheses are optional
const square = n => n * n;   // single-expression body: implicit return, no braces or "return" needed

console.log(square(5));   // 25
```

Arrow functions are a more compact syntax, introduced in 2015 alongside `let` / `const`. A single-expression arrow function (like `square` above) automatically returns that expression's value — no `{ }` braces or explicit `return` keyword needed, genuinely useful for short, simple functions.

> **MULTI-LINE ARROW FUNCTIONS NEED BRACES AND AN EXPLICIT RETURN**
>
> `const square = n => { return n * n; };` — once braces are added for multiple statements, the implicit return disappears, and `return` must be written explicitly, exactly like a normal function. Forgetting this is a genuinely common beginner mistake.

Default Parameters

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

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

Identical idea to PHP's default parameters from Fundamentals Chapter 5 — a fallback value used when the argument is omitted entirely.

Function vs Arrow Function — A Genuinely Important Difference: this

```
const counter = {
  count: 0,
  increment: function() {
    this.count++;   // "this" refers to the counter object — works correctly
    console.log(this.count);
  },
  incrementBroken: () => {
    this.count++;   // "this" does NOT refer to counter here — arrow functions don't have their own "thi
  }
};

counter.increment();   // 1 — works correctly
```

Regular functions get their own `this`, determined by how they're called. Arrow functions deliberately do **not** have their own `this` — they inherit it from their surrounding context instead. This matters significantly once objects (Chapter 7) and classes (Intermediate Chapter 6) are introduced; for now, the practical rule is: prefer regular `function` syntax for object methods, arrow functions for short standalone helpers and callbacks.

STYLE	HAS ITS OWN THIS?	IMPLICIT RETURN?	GOOD FOR
`function name() {}`	Yes	No	Object methods, general-purpose functions
`const f = function() {}`	Yes	No	Assigning a function to a variable, conditionally
`const f = () => {}`	No (inherits)	Only if single-expression, no braces	Short callbacks, array methods (Ch 6)

Scope — Mostly Like PHP, With One New Wrinkle

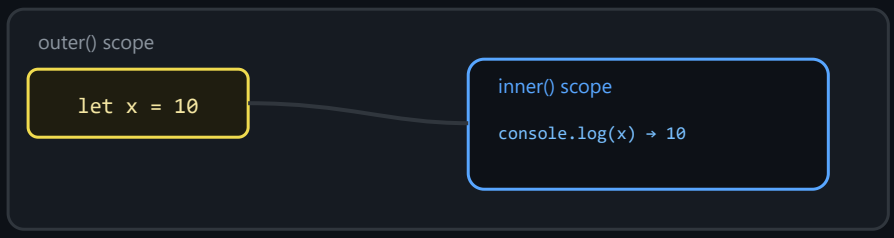
```
function outer() {
  let x = 10;   // local to this function, like PHP Fundamentals Chapter 5

  function inner() {
    console.log(x);   // 10 — inner functions CAN see variables from their enclosing function
  }

  inner();
}

outer();
```

A nested function can read variables from the function it's defined inside — this is new compared to PHP, where functions are never nested this way. The outer function's variables remain inaccessible from completely outside, exactly like PHP — but a function defined *inside* another one has visibility into its parent's scope. This "remembering" behaviour is called a **closure**, covered in full depth in Intermediate Chapter 2; this chapter just introduces that nested functions can see outer variables.



inner() can read outer()'s x, since it's defined inside outer's scope — visibility flows inward, not outward

Coding Challenges

CHALLENGE 1

Write the same function — isEven(number), returning true/false — three different ways: as a function declaration, as a function expression assigned to a const, and as an arrow function. Test all three with the same input and confirm they all produce identical results.

View solution

CHALLENGE 2

Write an arrow function calculateArea that takes width and height (with height defaulting to width, so it also works for squares with one argument) and returns their product using an implicit return. Call it three different ways and console.log() each result.

View solution

CHALLENGE 3

Write a function makeMultiplier(factor) that returns a NEW function — one that takes a number and multiplies it by factor. Use it to create double (factor 2) and triple (factor 3), then test both. Explain in a comment why the inner function can still access factor after makeMultiplier has already finished running.

View solution

CHAPTER 5 QUICK REFERENCE

- **function name() {}** — declaration; familiar, has its own this
- **const f = function() {}** — expression; function as a value assigned to a variable
- **const f = () => {}** — arrow function; no own this, implicit return for single expressions
- **Default parameters:** function f(x = "default")
- **this differs** between regular functions (own this) and arrow functions (inherited) — matters for object methods
- **Nested functions see outer variables** — visibility flows inward; this is the basis of closures (Intermediate Ch 2)
- **Functions are values** — "first-class functions," the foundation for array methods in Chapter 6
- **Next chapter:** arrays — methods like map, filter, forEach, reduce

CHAPTER 6 OF 10

Arrays

COURSE 1 · CH 6

Arrays: Creation, Indexing, and Core Methods

Storing ordered collections of values, and the four methods that do most of the real work: forEach, map, filter, reduce

Chapter 5 established that functions are values that can be passed around. This chapter puts that fact to immediate use: JavaScript arrays come with built-in methods that take a function as an argument and run it against every element. Mastering these four methods replaces almost all manual loop-writing for everyday array work.

Creating and Indexing Arrays

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

console.log(fruits[0]);        // "apple" — indexing starts at 0
console.log(fruits.length);    // 3
console.log(fruits[fruits.length - 1]);   // "cherry" — last element
```

An array is declared with `const` just like any other value — the array's *contents* can still change even though the binding itself can't be reassigned, the same rule that applied to objects would apply here too. Square brackets create the array; square brackets with an index read from it.

Adding and Removing Elements

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

numbers.push(4);       // [1, 2, 3, 4] — adds to the end
numbers.pop();         // [1, 2, 3] — removes from the end
numbers.unshift(0);    // [0, 1, 2, 3] — adds to the start
numbers.shift();       // [1, 2, 3] — removes from the start
```

`push` / `pop` work at the end of the array and are fast; `unshift` / `shift` work at the start and have to renumber every other element, so they're slower on large arrays.

forEach — Run a Function for Every Element

```
const scores = [72, 88, 91];

scores.forEach(score => {
  console.log(`Score: ${score}`);
});
// Score: 72
// Score: 88
// Score: 91
```

`forEach` takes a function and calls it once per element, passing that element in as the argument. It's a direct replacement for a `for` loop (Chapter 4) when the only goal is "do something with each item" — it doesn't build a new array or return a useful value.

map — Transform Every Element Into a New Array

```
const prices = [10, 20, 30];

const withTax = prices.map(price => price * 1.2);

console.log(withTax);   // [12, 24, 36]
console.log(prices);     // [10, 20, 30] — original array is untouched
```

`map` runs a function against every element and collects the return values into a brand-new array, leaving the original unchanged. Use `map` whenever the goal is "the same number of items, but transformed" — converting, scaling, formatting, and so on.

filter — Keep Only the Elements That Pass a Test

```
const ages = [15, 22, 17, 30];

const adults = ages.filter(age => age >= 18);

console.log(adults);   // [22, 30]
```

`filter` runs a function that returns `true` or `false` for each element, and keeps only the elements where it returned `true`. The result is a new, possibly shorter, array — the original is left alone, same as `map`.

reduce — Combine Every Element Into a Single Value

```
const cart = [9.99, 14.50, 3.25];

const total = cart.reduce((runningTotal, price) => runningTotal + price, 0);

console.log(total);   // 27.74
```

`reduce` is the least intuitive of the four, and the most powerful. Its function takes two arguments — the running result so far (**accumulator**) and the current element — and returns the new running result. The `0` after the function is the **starting value** for the accumulator, used before any element has been processed.

> **FORGETTING THE STARTING VALUE CHANGES THE RESULT**
>
> Without that second argument to `reduce`, the first array element is used as the starting accumulator instead, and the callback runs one fewer time. For sums starting at `0` this often still works by accident — but it silently breaks for anything where the first element isn't a safe starting point (string concatenation, building an object, etc.). Always pass the starting value explicitly.

Chaining Methods Together

```
const orders = [5, 12, 8, 20, 3];

const totalOfLargeOrders = orders
  .filter(qty => qty > 5)
  .map(qty => qty * 2)
  .reduce((sum, qty) => sum + qty, 0);

console.log(totalOfLargeOrders);   // 80
```

Because `filter` and `map` each return a new array, their results can be chained directly into the next method call. This reads almost like a sentence: "filter for orders over 5, double each one, then sum them up" — and is the idiomatic JavaScript style for this kind of data processing.

METHOD	RETURNS	USE IT WHEN...
forEach	undefined	You just need to do something with each item (e.g. log it)
map	New array, same length	You need a transformed version of every element
filter	New array, same or shorter	You need only the elements that match a condition
reduce	A single value	You need to combine everything into one result (sum, total, object)

Coding Challenges

CHALLENGE 1

Given const temps = [18, 22, 15, 30, 27], use map to create a new array converting each Celsius value to Fahrenheit (formula: C * 9/5 + 32). console.log() both the original array and the new one, confirming the original is unchanged.

View solution

CHALLENGE 2

Given const words = ["cat", "elephant", "dog", "hippopotamus", "ant"], use filter to create a new array containing only the words with more than 3 letters. console.log() the result.

View solution

CHALLENGE 3

Given const cart = [{ name: "Book", price: 12 }, { name: "Pen", price: 2 }, { name: "Bag", price: 25 }], use reduce to calculate the total price of all items. Then chain filter and reduce together to calculate the total price of only items costing more than 5.

View solution

CHAPTER 6 QUICK REFERENCE

- **Indexing:** arrays start at index 0; array.length gives the count
- **push/pop** — add/remove at the end (fast); **unshift/shift** — add/remove at the start (slower)
- **forEach** — run a function per element, returns nothing
- **map** — transform every element, returns a new array of the same length
- **filter** — keep elements passing a test, returns a new (possibly shorter) array
- **reduce** — combine all elements into a single value; always pass a starting value
- **map/filter/reduce never mutate** the original array — they return new data
- **Methods can be chained** — filter().map().reduce() reads like a processing pipeline
- **Next chapter:** objects — properties, methods, and the dot/bracket access patterns

CHAPTER 7 OF 10

Objects

COURSE 1 · CH 7

Objects: Properties, Methods, and Dot/Bracket Access

Grouping related data and behaviour together — the other major data structure alongside arrays

Chapter 6 covered arrays — ordered lists of values. Objects solve a different problem: grouping *named* pieces of related data together. Chapter 5's `counter` example already used one briefly; this chapter covers objects properly, including how methods and `this` actually work.

Creating an Object and Reading Properties

```
const person = {
  name: "Philip",
  age: 34,
  isStudent: true
};

console.log(person.name);    // "Philip" — dot notation
console.log(person["age"]);  // 34 — bracket notation
```

An object is a set of **key: value** pairs, wrapped in curly braces. Each key (also called a property name) maps to a value, which can be of any type — string, number, boolean, even another object or array.

Dot Notation vs Bracket Notation

```
const field = "age";

console.log(person.field);     // undefined — looks for a literal property called "field"
console.log(person[field]);    // 34 — looks up the property named by the variable's VALUE
```

Dot notation (`person.name`) is shorter and used by default. Bracket notation (`person["name"]`) is required whenever the property name is stored in a variable, contains spaces, or is computed at runtime — dot notation cannot do any of that.

PERSON.FIELD VS PERSON[FIELD] ARE NOT THE SAME THING

`person.field` always looks for a property literally named `field`. `person[field]` looks up whatever string is currently stored in the `field` variable. Mixing these up is a common source of silent `undefined` results.

Adding, Updating, and Deleting Properties

```
person.city = "London";     // adds a new property
person.age = 35;             // updates an existing one
delete person.isStudent;    // removes a property entirely

console.log(person);   // { name: "Philip", age: 35, city: "London" }
```

Properties can be added or changed after creation simply by assigning to them — even though `person` is declared with `const`, exactly the same rule that allowed array contents to change in Chapter 6. `const` only locks the variable binding itself, never the object's internal contents.

Methods — Functions Stored as Properties

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

person.greet();   // "Hi, I'm Philip"
```

A property whose value is a function is called a **method**. Inside a method, `this` refers to the object the method was called on — `person` in this case — which is how `greet` can reach `person.name` without naming `person` directly. This is exactly the regular-function behaviour flagged in Chapter 5: methods should use the `function` keyword, not arrow syntax, specifically so `this` works.

Nested Objects and Arrays of Objects

```
const user = {
  name: "Philip",
  address: { city: "London", postcode: "E1 6AN" }
};

console.log(user.address.city);   // "London" — chain dots to go deeper

const products = [
  { name: "Book", price: 12 },
  { name: "Pen", price: 2 }
];

console.log(products[0].name);   // "Book"
```

Objects and arrays nest freely inside each other. `products` here is the same array-of-objects shape used in Chapter 6's cart challenge — it's worth recognising this pattern, since it's extremely common for representing real-world lists of records (rows from a database, items in a cart, entries in a form).

Looping Over an Object's Properties

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

for (const subject in scores) {
  console.log(`${subject}: ${scores[subject]}`);
}
// maths: 88
// science: 72
// art: 95
```

`for...in` (introduced in Chapter 4 for arrays, where it's best avoided) is the natural fit for objects — it loops over each property *name*, which can then be used with bracket notation to read the matching value. Notice bracket notation is required here, since `subject` is a variable holding the property name.

> **OBJECT.KEYS, OBJECT.VALUES, OBJECT.ENTRIES**
>
> `Object.keys(scores)` returns `["maths", "science", "art"]` as a real array — meaning `map` / `filter` / `reduce` from Chapter 6 can then be used on an object's data. This combination is covered properly in Intermediate Chapter 1.

Coding Challenges

CHALLENGE 1

Create an object book with properties title, author, and pages. Log each property using dot notation, then log the same three values again using bracket notation with a variable holding the property name.

View solution

CHALLENGE 2

Create an object car with properties make, model, and a method describe() that uses this to log a sentence combining make and model. Call describe(), then add a new property year to car afterwards and call describe() again, updating it to include the year.

View solution

CHALLENGE 3

Create an object inventory where each key is an item name and each value is a quantity (e.g. apples: 10, bananas: 5). Use a for...in loop to log every item with its quantity, and keep a running total of all quantities combined, logging the total at the end.

View solution

CHAPTER 7 QUICK REFERENCE

- **Object literal:** { key: value, key2: value2 }
- **Dot notation** (obj.key) — short, but key must be a literal name
- **Bracket notation** (obj[expr]) — required for variable/dynamic/spaced keys
- **const objects are still mutable** — properties can be added, changed, or deleted
- **Methods** are functions stored as properties; use this inside them to reference the object
- **Use function, not arrow syntax,** for methods — arrow functions don't get their own this (Ch 5)
- **for...in** loops over an object's property names
- **Next chapter:** the DOM — selecting and manipulating real HTML elements with JavaScript

CHAPTER 8 OF 10

The DOM

COURSE 1 · CH 8

The DOM: Selecting and Manipulating HTML Elements

Where JavaScript stops being abstract — reaching into a real page and changing what's actually on screen

Every chapter so far has run entirely in the console. The **DOM** (Document Object Model) is the browser's live, in-memory representation of the HTML page — and JavaScript can read and change it directly, which is how a page updates without a reload. This chapter is the bridge between "writing JavaScript" and "JavaScript that actually does something visible."

Selecting a Single Element

```
const heading = document.querySelector("h1");
const box = document.querySelector("#intro");
const firstItem = document.querySelector(".list-item");

console.log(heading);
```

`document.querySelector()` takes a CSS selector — the same syntax used in a stylesheet — and returns the **first** matching element on the page, or `null` if nothing matches. `#intro` selects by ID, `.list-item` by class, exactly as in CSS.

Selecting Multiple Elements

```
const items = document.querySelectorAll(".list-item");

console.log(items.length);   // however many .list-item elements exist

items.forEach(item => {
  console.log(item.textContent);
});
```

`querySelectorAll()` returns **every** matching element as a NodeList — array-like enough that `forEach` from Chapter 6 works directly on it, even though it's technically not a true array.

QUERYSELECTOR VS QUERYSELECTORALL

Use `querySelector` when exactly one element is expected (often by ID). Use `querySelectorAll` whenever there could be several matches (a class shared by many elements) and all of them need handling.

Reading and Changing Text Content

```
const heading = document.querySelector("h1");

console.log(heading.textContent);   // reads the current text
heading.textContent = "Updated by JavaScript";   // changes it on the page, instantly
```

`textContent` is a property on the element, just like the object properties from Chapter 7 — reading it returns the current text, and assigning to it updates the page immediately, with no reload required.

Changing Styles and Classes

```
box.style.color = "red";            // inline style, one property at a time
box.style.backgroundColor = "#222";

box.classList.add("highlighted");     // adds a CSS class
box.classList.remove("highlighted");  // removes it
box.classList.toggle("highlighted");  // adds if missing, removes if present
```

`element.style` sets individual CSS properties directly (camelCase instead of CSS's hyphenated names — `backgroundColor`, not `background-color`). `classList` is almost always the better choice for anything beyond a one-off tweak, since it keeps styling rules in CSS where they belong and JavaScript just toggles which rules apply.

Responding to Events

```
const button = document.querySelector("#myButton");

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

`addEventListener` takes an event name (`"click"`, `"input"`, `"submit"`, and many others) and a function to run when that event happens. This is the first genuinely common, practical use for the function-as-a-value behaviour from Chapter 5 — the function itself is handed over and the browser calls it later, whenever the click actually occurs.

RUN YOUR SCRIPT AFTER THE PAGE HAS LOADED

If a `<script>` runs before the HTML below it exists yet, `querySelector` returns `null` and calling a method on it throws an error. Either place scripts at the end of `<body>`, or wrap the code in `document.addEventListener("DOMContentLoaded", () => { ... })`.

Creating New Elements

```
const list = document.querySelector("#myList");

const newItem = document.createElement("li");
newItem.textContent = "A brand new item";

list.appendChild(newItem);   // inserts it at the end of the list
```

`createElement` builds a new element entirely in memory — it doesn't appear on the page until it's attached somewhere with `appendChild` (or a similar method). This pattern — combined with an array of data and a loop — is how real pages render dynamic lists.

TASK	METHOD/PROPERTY
Select one element	`document.querySelector(selector)`
Select all matching elements	`document.querySelectorAll(selector)`
Read/change text	`element.textContent`
Read/change one CSS property	`element.style.property`
Add/remove/toggle a CSS class	`element.classList.add/remove/toggle()`
Run code on an event	`element.addEventListener(event, fn)`
Build a new element	`document.createElement(tag)`

Coding Challenges

CHALLENGE 1

Assume the page has a

Hello

. Select it with querySelector, log its current textContent, then change its text to "Updated!" and change its color to blue using element.style.

View solution

CHALLENGE 2

Assume the page has a button with id toggleBtn and a div with id panel. Add a click event listener to the button that toggles a class called "open" on panel each time it's clicked, using classList.toggle.

View solution

CHALLENGE 3

Given const fruits = ["Apple", "Banana", "Cherry"] and an empty

on the page, write code that creates a new li element for each fruit (using createElement and textContent) and appends each one to fruitList.

View solution

CHAPTER 8 QUICK REFERENCE

- **querySelector(selector)** — first matching element, or null
- **querySelectorAll(selector)** — all matching elements, as a NodeList (forEach works on it)
- **element.textContent** — read or change an element's text
- **element.style.property** — set one CSS property directly (camelCase)
- **element.classList.add/remove/toggle()** — preferred way to change appearance via CSS classes
- **element.addEventListener(event, fn)** — run a function when something happens (click, input, etc.)
- **document.createElement(tag) + appendChild()** — build and insert new elements dynamically
- **Run scripts after the HTML exists** — end of body, or inside a DOMContentLoaded listener
- **Next chapter:** forms and user input — reading values, validating, and responding to submission

CHAPTER 9 OF 10

Forms and User Input

COURSE 1 · CH 9

Forms and User Input

Reading what a visitor typed, checking it's valid, and reacting when they submit

Chapter 8 covered reaching into the page and reacting to clicks. The most common real-world use of that is forms — text fields, checkboxes, dropdowns — where a page needs to read what the user typed, check it, and respond. This chapter ties together `querySelector`, `addEventListener`, and the conditional logic from Chapter 3 into one practical workflow.

Reading an Input's Value

```
const nameInput = document.querySelector("#nameInput");

console.log(nameInput.value);   // whatever the user has currently typed, as a string
```

Unlike most elements, a form input's current text lives in its `value` property — not `textContent`. `value` is always a string, even for a number input, which matters when doing arithmetic with it.

Listening for Input Changes

```
nameInput.addEventListener("input", () => {
  console.log(`Current value: ${nameInput.value}`);
});
```

The `"input"` event fires every time the field's value changes — every keystroke, paste, or autofill — making it the right choice for live feedback like character counters or instant validation messages, as opposed to `"change"`, which only fires once the field loses focus.

Handling Form Submission

```
const form = document.querySelector("#signupForm");

form.addEventListener("submit", (event) => {
  event.preventDefault();   // stops the page from reloading
  console.log("Form submitted!");
});
```

A form's default behaviour on submit is to reload the page and send its data to a server — almost never what's wanted when JavaScript is handling things instead. `event.preventDefault()` stops that default behaviour, while still letting the rest of the handler run normally.

> **FORGETTING PREVENTDEFAULT() LOOKS LIKE THE SCRIPT "DID NOTHING"**
> Without it, the page reloads the instant submit happens, wiping out the console and any in-memory state before there's time to even notice the handler ran. If a submit handler appears to silently fail, this is the first thing to check.

Validating Input Before Accepting It

```
form.addEventListener("submit", (event) => {
  event.preventDefault();

  const name = document.querySelector("#nameInput").value.trim();
  const errorBox = document.querySelector("#errorBox");

  if (name === "") {
    errorBox.textContent = "Name is required.";
    return;
  }

  errorBox.textContent = "";
  console.log(`Welcome, ${name}!`);
});
```

`.trim()` removes leading/trailing whitespace, so a field containing only spaces is correctly treated as empty. Using `return` inside the handler stops execution immediately when validation fails, the same early-exit pattern functions have used since Chapter 5 — nothing after it runs until the user fixes the problem and resubmits.

Checkboxes and Select Dropdowns

```
const subscribe = document.querySelector("#subscribeCheckbox");
console.log(subscribe.checked);   // true or false — NOT value

const country = document.querySelector("#countrySelect");
console.log(country.value);   // the value of the currently selected
```

Checkboxes use `checked` (a boolean) rather than `value` for their state. Dropdowns (`<select>`) use `value`, same as text inputs, returning whichever `<option>`'s value attribute is currently selected.

ELEMENT	READ STATE WITH	USEFUL EVENT
Text input	`input.value`	"input" (live) or "change" (on blur)
Checkbox	`checkbox.checked`	"change"
Select dropdown	`select.value`	"change"
Whole form	—	"submit" (always call event.preventDefault())

Coding Challenges

CHALLENGE 1

Assume an input with id ageInput and a p with id ageOutput. Add an "input" event listener to ageInput that updates ageOutput's textContent live to say "You typed: X" every time the value changes.

View solution

CHALLENGE 2

Assume a form with id loginForm containing an input #username and a p#errorBox. On submit, prevent the default reload, trim the username, and if it's empty show "Username is required." in errorBox; otherwise clear errorBox and log a welcome message.

View solution

CHALLENGE 3

Assume a checkbox #agreeCheckbox and a button #submitBtn that starts disabled. Add a "change" listener to the checkbox that enables submitBtn (button.disabled = false) when checked, and disables it again (button.disabled = true) when unchecked.

View solution

CHAPTER 9 QUICK REFERENCE

- **input.value** — current text in an input/textarea/select (always a string)
- **checkbox.checked** — boolean state of a checkbox, not value
- **"input"** — fires on every keystroke; **"change"** — fires once focus leaves the field
- **form.addEventListener("submit", ...)** — always call event.preventDefault() first
- **.trim()** — strips whitespace before checking if a value is "empty"
- **return inside a handler** — early-exit pattern for stopping on invalid input
- **element.disabled = true/false** — enable/disable form controls dynamically
- **Next chapter:** asynchronous JavaScript — fetch, promises, and async/await

CHAPTER 10 OF 10

Asynchronous JavaScript

COURSE 1 · CH 10

Asynchronous JavaScript: Promises, async/await, and fetch

Dealing with things that take time — network requests, timers, and anything that doesn't finish instantly

Every example so far has run top to bottom, instantly. Real pages constantly wait on things that take time — loading data from a server being the most common. JavaScript handles this with **promises**, and the modern, readable way to work with them: `async` / `await`. This is the final Fundamentals chapter, and it pulls together functions (Ch 5), objects (Ch 7), and the DOM (Ch 8–9) into one realistic workflow.

The Problem: JavaScript Doesn't Wait

```
console.log("1. Starting");

setTimeout(() => {
  console.log("2. This runs later");
}, 1000);

console.log("3. This runs before #2!");

// Logged order: 1, 3, 2 — NOT 1, 2, 3
```

`setTimeout` schedules a function to run later without pausing anything else. JavaScript moves straight on to the next line rather than waiting — this is what "asynchronous" means: code that will run eventually, not necessarily in the order it's written.

Promises — A Placeholder for a Future Value

```
const waitOneSecond = new Promise((resolve) => {
  setTimeout(() => {
    resolve("Done waiting!");
  }, 1000);
});

waitOneSecond.then((result) => {
  console.log(result);   // "Done waiting!" — logged after ~1 second
});
```

A **Promise** represents a value that doesn't exist yet, but will at some point — either successfully (`resolve`) or with an error (`reject`). `.then()` registers a function to run once the promise resolves, receiving the resolved value as its argument.

async/await — Promises Without the .then() Chains

```
async function run() {
  console.log("Before waiting");
  const result = await waitOneSecond;
  console.log(result);    // "Done waiting!"
  console.log("After waiting");
}

run();
```

`async` before a function lets `await` be used inside it. `await` pauses *that function* (not the whole page) until the promise resolves, then continues with the resolved value — reading top-to-bottom almost exactly like the synchronous code from every earlier chapter, while still being genuinely non-blocking.

> **AWAIT ONLY WORKS INSIDE AN ASYNC FUNCTION**
>
> Trying to use `await` in a normal function is a syntax error. This is the one new keyword pairing to remember: no `async` on the function, no `await` inside it.

fetch — Getting Real Data from a Server

```
async function getUser() {
  const response = await fetch("https://api.example.com/user/1");
  const data = await response.json();

  console.log(data.name);   // the object's "name" property, from Chapter 7
}

getUser();
```

`fetch()` requests a URL and returns a promise. The first `await` gets the response itself; `response.json()` then parses the body text into a real JavaScript object — another promise, needing its own `await` — at which point Chapter 7's dot/bracket notation works on it normally.

Handling Errors with try/catch

```
async function getUser() {
  try {
    const response = await fetch("https://api.example.com/user/1");
    const data = await response.json();
    console.log(data.name);
  } catch (error) {
    console.log("Something went wrong:", error.message);
  }
}
```

A network request can fail — no connection, server error, invalid URL. Wrapping `await` calls in `try/catch` means a failure jumps straight to the `catch` block instead of leaving an unhandled error. Without this, a failed request can silently break the rest of the function with no clear feedback to the user.

> **A NON-200 RESPONSE DOESN'T AUTOMATICALLY THROW**
>
> `fetch` only rejects (triggering `catch`) on a genuine network failure — a 404 or 500 response is still considered a "successful" fetch as far as the promise is concerned. Check `response.ok` (a boolean) explicitly if a non-success status code needs separate handling.

Putting It Together: Fetch + DOM

```
async function loadUser() {
  const output = document.querySelector("#userOutput");
  output.textContent = "Loading...";

  try {
    const response = await fetch("https://api.example.com/user/1");
    const user = await response.json();
    output.textContent = `Hello, ${user.name}!`;
  } catch (error) {
    output.textContent = "Failed to load user.";
  }
}
```

This is the complete, realistic pattern: show a loading state immediately, fetch data, then update the page with the result — or an error message if it fails. Every chapter since Chapter 5 contributes something here: functions, objects, DOM updates, and now asynchronous waiting.

Coding Challenges

CHALLENGE 1

Write an async function delayedGreet(name) that uses await with a Promise wrapping setTimeout (1 second) to wait, then console.logs `Hello, ${name}!`. Call it and log "Calling..." immediately before, to confirm the greeting really does log after a delay.

View solution

CHALLENGE 2

Write an async function getPost(id) that fetches from `https://jsonplaceholder.typicode.com/posts/${id}`, parses the JSON, and logs the post's title. Wrap it in try/catch and log "Failed to fetch post." on any error.

View solution

CHALLENGE 3

Assume a button #loadBtn and a p#status. Add a click listener that's an async function: on click, set status text to "Loading...", await a fetch to `https://jsonplaceholder.typicode.com/users/1`, then set status to the user's email — or "Error loading data." if the fetch fails.

View solution

CHAPTER 10 QUICK REFERENCE

- **Asynchronous** code doesn't block — JavaScript moves on while it's pending
- **Promise** — a placeholder for a value that resolves (success) or rejects (failure) later
- **async function** — required wrapper to use await inside
- **await** — pauses the async function (not the page) until a promise settles
- **fetch(url)** — returns a promise for an HTTP response
- **response.json()** — parses the response body into a usable object/array
- **try/catch** — wraps await calls to handle network/parsing failures gracefully
- **response.ok** — check explicitly for non-200 status codes; fetch won't throw on its own for those
- **This completes JavaScript Fundamentals.** Intermediate begins with closures, the spread/rest operators, and ES6 classes.