JavaScript
From Beginner to Advanced

A Complete Concept-by-Concept Guide with Hands-On Exercises

Prepared by

ZOlaXTech

zolaxtech.com.et

Written and practiced entirely in Visual Studio Code

August 2026

How to Use This Guide

This guide continues directly from the zolaxtech.com.et HTML5 and CSS guides and takes you from zero interactivity to a fully dynamic, validated, data-driven site, using Visual Studio Code throughout. Every concept is presented the same way, so you always know what to expect:

- **The concept or feature:** shown in a monospace font, e.g. addEventListener().
- **What it does:** a plain-language explanation of its purpose and behavior.
- **Values / Details:** a table of the concept's most important variations, parameters, or values and exactly what each one does.
- **Example:** a real, working code snippet you can type directly into VS Code's script.js.
- **Exercise:** a short hands-on task, using zolaxtech.com.et as the running example, so you practice each concept immediately instead of just reading about it.

Each chapter ends with a larger "Chapter Project" exercise that combines everything covered in that chapter into one realistic piece of the Zolax Tech website. By the final chapter, you will have added working forms validation, interactive menus, dynamic DOM updates, live data fetched from a server, and clean, well-organized, error-handled JavaScript to a real multi-page business website built and previewed entirely inside Visual Studio Code.

Table of Contents

Chapter 1: Getting Started with JavaScript in Visual Studio Code

JavaScript is the programming language that makes web pages interactive validating forms, responding to clicks, loading data without a page refresh. This chapter sets up your workspace so you can write, run, and debug JavaScript immediately, continuing the zolaxtech.com.et project from the HTML and CSS guides.

Step 1: Reuse your project folder

Open the same zolaxtech-site folder in VS Code. You should already have index.html and styles.css from the earlier guides.

Step 2: Create your first script file

In the Explorer panel, create a new file named script.js in the same folder as index.html.

Step 3: Link the script to your HTML

Just before the closing </body> tag in index.html, add <script src="script.js" defer></script>. Placing it at the end (or using defer) ensures the page's HTML loads before your JavaScript tries to interact with it.

Step 4: Open the browser console

With Live Server running, open your page in the browser, right-click anywhere and choose "Inspect", then click the "Console" tab. This is where JavaScript's output and errors appear you will use it constantly.

Step 5: Write your first line

In script.js, type console.log ("Zolax Tech script loaded"); and save. Switch to the browser console and confirm the message appears this confirms your file is linked and running correctly.

Step 6: Install a linting extension

Search Extensions for "ESLint" and install it. It underlines likely mistakes (like a missing semicolon or an undefined variable) directly in your editor as you type, before you even run the code.

Step 7: Use the integrated terminal for Node.js (optional)

Open a terminal with Ctrl+` and type node -v. If Node.js is installed, this shows a version number, meaning you can also run plain .js files outside the browser with node script.js useful for practicing logic in isolation.

Chapter Summary Project

🏁 Chapter Project

Link script.js to index.html with the defer attribute, write three separate console.log() statements introducing yourself and your goal of learning JavaScript for the Zolax Tech site, save, and confirm all three lines appear in the correct order in the browser console.

Chapter 2: Syntax Basics, Variables & Data Types

This chapter covers the fundamental building blocks: how to store information in variables and the different kinds of values JavaScript can work with.

let

Declares a variable that CAN be reassigned later, and is scoped to the nearest enclosing block ({}) the standard way to declare a variable whose value will change.

Example

```
let visitorCount = 0;
visitorCount = visitorCount + 1;
console.log(visitorCount); // 1
```

> **➖ Exercise**
> Declare a let variable named quoteRequests set to 0, then write two lines that each increase it by 1, logging the value after each increase.

const

Declares a variable that CANNOT be reassigned after its initial value is set. Use const by default for anything that won't change it makes code easier to reason about and prevents accidental reassignment bugs.

Example

```
const companyName = "Zolax Tech";
const foundedYear = 2020;
// companyName = "Other"; // this would throw an error
```

> **➖ Exercise**
> Declare const variables for companyName and companyLocation with appropriate string values, and log a sentence combining both.

var (legacy)

The original way to declare variables before let/const existed in 2015. Scoped to the whole function rather than the block, which causes confusing bugs modern code should avoid var entirely in favor of let and const.

Example

```
// Old style (avoid in new code):
var x = 10;

// Modern equivalent:
```

```
let y = 10;
```

> **➖ Exercise**
> Write a short code comment explaining, in your own words, why let is generally preferred over var in modern JavaScript.

String

Represents text, written inside single quotes, double quotes, or backticks (template literals).

Values / Details

Concept / Value	What it does
'single' or "double" quotes	Plain text; pick one style and stay consistent across your project.
`template literals`	Backtick-quoted strings that allow embedded expressions using ${ } and multi-line text directly, without concatenation.
Common methods	.length (number of characters), .toUpperCase(), .toLowerCase(), .trim(), .includes(), .slice().

Example

```
const service = "Web Development";
const price = 1500;
const message = `${service} costs ${price} ETB. `;
console.log(message.toUpperCase());
```

> **➖ Exercise**
> Create a template literal that combines a service name and price variable into one readable sentence, then log an uppercase version of it using .toUpperCase().

Number

Represents both integers and decimals JavaScript does not have separate types for whole numbers versus decimals the way some languages do.

Values / Details

Concept / Value	What it does
Arithmetic	Supports +, -, *, /, % (remainder), and ** (exponent).
Common methods	Number.isInteger(), .toFixed(2) (rounds to a fixed number of decimal places, returning a string).

Example

```
const basicPlanPrice = 500;
const discount = 0.1;
const finalPrice = basicPlanPrice - basicPlanPrice * discount;
console.log(finalPrice.toFixed(2)); // "450.00"
```

> **➼ Exercise**
> Calculate a 15% discount on a service price of your choosing using Number arithmetic, and log the final price formatted to two decimal places with .toFixed(2).

Boolean

Represents exactly one of two values: true or false the foundation of all conditional logic.

Example

```
const isOpenNow = true;
const hasDiscount = false;
console.log(isOpenNow, hasDiscount);
```

> **➼ Exercise**
> Declare two boolean variables representing whether Zolax Tech is currently accepting new clients and whether a promotion is active, and log both.

null & undefined

Two distinct 'empty' values. undefined means a variable has been declared but never given a value (JavaScript's own default); null means a developer has deliberately set a variable to represent 'no value' or 'empty on purpose'.

Example

```
let selectedPlan; // undefined not yet assigned
let cancelledOrder = null; // deliberately empty
console.log(selectedPlan, cancelledOrder);
```

> **➼ Exercise**
> Declare a variable without assigning it and log it to confirm it shows undefined, then explicitly set a second variable to null and log it as well, noting the difference in intent between the two.

Array

An ordered, numbered list of values, written inside square brackets. Covered fully in Chapter 6.

Example

```
const services = ["Web Development", "Mobile Apps", "UI/UX Design"];
console.log(services[0]); // "Web Development"
```

> **➖ Exercise**
> Create an array of four Zolax Tech services and log the second item using its index.

Object

A collection of related key-value pairs, written inside curly braces. Covered fully in Chapter 7.

Example

```
const client = {
  name: "Selam Traders",
  plan: "Pro",
  active: true,
};
console.log(client.name);
```

> **➖ Exercise**
> Create an object representing a Zolax Tech client with name, plan, and active properties, then log the plan property using dot notation.

typeof operator

Returns a string naming the data type of a given value extremely useful when debugging unexpected values.

Example

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

> **➖ Exercise**
> Log the typeof result for one variable of each type covered in this chapter (string, number, boolean, array, object) to confirm you can predict each result before running it.

Chapter Summary Project

🏁 Chapter Project

Build a small 'company facts' script for Zolax Tech using const and let appropriately: a company name (string), founding year (number), an isHiring flag (boolean), a services array, and a client object with at least three properties. Log a full sentence to the console that combines several of these values using a template literal.

Chapter 3: Operators & Expressions

Operators combine values to produce new ones this chapter covers arithmetic, comparison, logical, and assignment operators.

Arithmetic operators

Perform mathematical calculations on numbers.

Values / Details

Concept / Value	What it does
+ - * /	Addition, subtraction, multiplication, division.
%	Remainder (modulo) returns what's left over after division, e.g. 10 % 3 is 1. Commonly used to check odd/even (n % 2 === 0).
**	Exponentiation, e.g. 2 ** 3 is 8.
++ --	Increment/decrement a variable by exactly 1.

Example

```
let total = 500 + 1500;
let remainder = 10 % 3;
let squared = 4 ** 2;
let count = 0;
count++;
console.log(total, remainder, squared, count);
```

➖ Exercise

Calculate the total price of three Zolax Tech services added together, and use the % operator to check whether the total is an even number (total % 2 === 0).

Assignment operators

Assign a value to a variable, optionally combined with an arithmetic operation to update it based on its own current value.

Values / Details

Concept / Value	What it does
=	Basic assignment.
+= -= *= /=	Shorthand that combines the operation with reassignment, e.g. total += 10; is identical to total = total + 10;.

Example

```
let cartTotal = 500;
cartTotal += 1500; // now 2000
cartTotal -= 200; // now 1800
console.log(cartTotal);
```

> **➖ Exercise**
> Start with a cartTotal of 0 and use += three times to add three different service prices, logging the running total after each addition.

Comparison operators

Compare two values and produce a boolean result.

Values / Details

Concept / Value	What it does
=== !==	Strict equality/inequality compares BOTH value and type, with no automatic type conversion. Always prefer these over == and !=.
== !=	Loose equality/inequality converts types before comparing (e.g. "5" == 5 is true), which frequently causes confusing bugs. Avoid in new code.
> < >= <=	Greater than, less than, greater-or-equal, less-or-equal.

Example

```
console.log(5 === 5);   // true
console.log(5 === "5"); // false (different types)
console.log(5 == "5");  // true  (loose  avoid)
console.log(1500 > 500); // true
```

> **➖ Exercise**
> Write and log three strict-equality (===) comparisons involving your Zolax Tech variables from Chapter 2, and one comparison demonstrating why === is safer than == by comparing a number to the equivalent string.

Logical operators

Combine or invert boolean values, used constantly inside conditionals.

Values / Details

Concept / Value	What it does
&& (AND)	True only if BOTH sides are true.
|| (OR)	True if AT LEAST ONE side is true.
! (NOT)	Flips true to false and vice versa.
?? (nullish coalescing)	Returns the right-hand value only if the left-hand value is null or undefined (unlike ||, which also triggers on 0 or an empty string).

Example

```
const isOpen = true;
const hasStaffAvailable = false;
console.log(isOpen && hasStaffAvailable); // false
console.log(isOpen || hasStaffAvailable); // true
console.log(!isOpen);                     // false

let discount = null;
console.log(discount ?? 0); // 0, since discount is null
```

> **➖ Exercise**
> Write a logical expression that checks whether Zolax Tech should show a 'currently accepting clients' banner: true only when isOpen is true AND hasStaffAvailable is true. Then use ?? to provide a default value of 0 for a discount variable that starts as null.

Ternary operator

A compact one-line shorthand for a simple if/else, producing a value directly based on a condition.

Values / Details

Concept / Value	What it does
condition ? valueIfTrue : valueIfFalse	Evaluates the condition and returns one of two values.

Example

```
const stock = 0;
const status = stock > 0 ? "In Stock" : "Out of Stock";
console.log(status);
```

➖ Exercise

Use a ternary operator to set a message variable to 'Hiring' or 'Not Hiring' based on an isHiring boolean, and log the result.

String concatenation & template literals

Two ways to combine strings and variables into one piece of text.

Values / Details

Concept / Value	What it does
+ (concatenation)	Joins strings together manually, e.g. "Hello, " + name + "!" works, but gets unwieldy with many variables.
Template literals (recommended)	Backtick strings using ${expression} to embed variables directly, far more readable for anything beyond a trivial join.

Example

```
const name = "Selam";
// Old way:
console.log("Hello, " + name + "! Welcome to Zolax Tech.");
// Modern way:
console.log(`Hello, ${name}! Welcome to Zolax Tech.`);
```

➖ Exercise

Rewrite a string concatenation example using + into a template literal instead, confirming both produce identical output.

Chapter Summary Project

🏁 Chapter Project

Build a small 'quote calculator' script: define prices for three services as numbers, calculate their combined total using arithmetic operators, apply a 10% discount only if the total is greater than 1000 using a ternary operator, and log a full summary sentence using a template literal and strict equality checks to confirm your discount logic worked correctly.

Chapter 4: Control Flow

Control flow statements let your program make decisions and repeat actions the difference between a static script and real logic.

if / else if / else

Runs a block of code only when a condition is true, with optional additional conditions and a final fallback.

Example

```
const leadScore = 75;

if (leadScore >= 80) {
  console.log("Hot lead  contact immediately");
} else if (leadScore >= 50) {
  console.log("Warm lead  follow up this week");
} else {
  console.log("Cold lead  add to newsletter");
}
```

> **➖ Exercise**
>
> Write an if/else if/else chain that categorizes a project budget variable into 'Small Project' (under 5000 ETB), 'Medium Project' (5000-20000 ETB), or 'Large Project' (above 20000 ETB), and log the resulting category.

switch

Compares one value against several possible exact matches often more readable than a long if/else if chain when checking a single variable against many specific values.

Values / Details

Concept / Value	What it does
case value:	Runs the matching block.
break;	Stops execution from 'falling through' into the next case easy to forget, and a common source of bugs.
default:	Runs if no case matches, similar to a final else.

Example

```
const plan = "pro";

switch (plan) {
  case "basic":
    console.log("500 ETB/month");
    break;
  case "pro":
```

```
    console.log("1500 ETB/month");
    break;
  case "enterprise":
    console.log("Contact us for pricing");
    break;
  default:
    console.log("Unknown plan");
}
```

> **➖ Exercise**
> Write a switch statement that logs a different service description depending on a serviceType variable set to "web", "mobile", or "design", including a default case for anything else.

for loop

Repeats a block of code a specific number of times, using a counter variable you fully control the most common loop for 'do this N time's or looping through an array by index.

Values / Details

Concept / Value	What it does
for (start; condition; step) { }	Runs an initializer once, checks the condition before each pass, and runs the step after each pass.

Example

```
const services = ["Web Development", "Mobile Apps", "UI/UX Design"];

for (let i = 0; i < services.length; i++) {
  console.log(`${i + 1}. ${services[i]}`);
}
```

> **➖ Exercise**
> Use a classic for loop to log every item in your services array from Chapter 2, numbered starting at 1 instead of 0.

while loop

Repeats a block of code for as long as a condition remains true, checked BEFORE each pass used when you don't know in advance exactly how many times you'll need to loop.

Example

```
let ticketsRemaining = 3;

while (ticketsRemaining > 0) {
```

```
  console.log (`Processing ticket... ${ticketsRemaining} left`);
  ticketsRemaining--;
}
```

> **➖ Exercise**
> Simulate processing a queue of 5 support tickets using a while loop that logs a message each iteration and decreases a counter until it reaches 0.

do...while loop

Identical to while, except the condition is checked AFTER each pass guaranteeing the block runs at least once, even if the condition starts out false.

Example

```
let attempts = 0;

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

> **➖ Exercise**
> Write a do...while loop that always logs at least one 'Attempting connection...' message, even if a hypothetical maxAttempts variable starts at 0, to demonstrate the guaranteed-first-run behavior.

for...of loop

A modern, simpler way to loop directly over the VALUES of an array (or any iterable), without needing to manage an index counter manually preferred over a classic for loop whenever you don't need the index itself.

Example

```
const services = ["Web Development", "Mobile Apps", "UI/UX Design"];

for (const service of services) {
  console.log(service);
}
```

> **➖ Exercise**
> Rewrite your Chapter 4 for-loop exercise using for...of instead, and compare how much simpler the syntax is when you don't need the index.

for...in loop

Loops over the KEYS (property names) of an object not typically used for arrays, where for...of or array methods are preferred instead.

Example

```
const client = {name: "Selam Traders", plan: "Pro", active: true};

for (const key in client) {
  console.log(`${key}: ${client[key]}`);
}
```

> **➖ Exercise**
> Use a for...in loop to log every property name and value of your client object from Chapter 2.

break & continue

Two ways to alter a loop's normal flow from the inside.

Values / Details

Concept / Value	What it does
break	Immediately exits the loop entirely, skipping any remaining iterations.
continue	Skips the REST of the current iteration only, and moves on to the next one.

Example

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

for (const n of numbers) {
  if (n === 4) break;   // stop entirely at 4
  if (n % 2 !== 0) continue; // skip odd numbers
  console.log(n); // logs 2
}
```

> **➖ Exercise**
> Loop through an array of numbers 1 through 10, using continue to skip any number divisible by 3, and break to stop the loop entirely once you reach a number greater than 8.

Chapter Summary Project

🏁 Chapter Project

Build a 'client intake' script: create an array of five fictional lead scores (numbers), loop through them with for...of, and use if/else if/else inside the loop to log each lead as Hot, Warm, or Cold. Separately, use a switch statement to print a pricing line for three different plan variables ("basic", "pro", "enterprise"), and use break inside a loop to stop processing as soon as you find the first Hot lead.

Chapter 5: Functions

Functions package up reusable blocks of logic. This chapter covers every way to define one and the scoping rules that govern how they see variables.

Function declaration

The classic way to define a named, reusable function. Function declarations are 'hoisted' they can be called from code written ABOVE where they're defined in the file, unlike most other JavaScript.

Example

```
function calculateTotal(price, quantity) {
  return price * quantity;
}

console.log(calculateTotal(500, 3)); // 1500
```

➖ Exercise

Write a function declaration named calculateDiscount that takes a price and a discountPercent, and returns the discounted price.

Parameters, arguments & return

Parameters are the named placeholders a function expects; arguments are the actual values passed in when calling it; return sends a value back out of the function to wherever it was called (a function with no return implicitly returns undefined).

Values / Details

Concept / Value	What it does
Default parameters	function greet(name = "Guest") lets a parameter fall back to a default value if no argument is passed for it.

Example

```
function greetClient(name = "Guest") {
  return `Welcome to Zolax Tech, ${name}!`;
}

console.log(greetClient("Selam")); // "Welcome to Zolax Tech, Selam!"
console.log(greetClient());        // "Welcome to Zolax Tech, Guest!"
```

> **➖ Exercise**
> Write a greetClient function with a default parameter, and call it twice once with a name argument, once without to confirm the default kicks in correctly.

Function expression

Defines a function and assigns it to a variable. Unlike declarations, function expressions are NOT hoisted they can only be called after the line where they're defined.

Example

```
const calculateTax = function (amount) {
  return amount * 0.15;
};

console.log(calculateTax(1000)); // 150
```

> **➖ Exercise**
> Rewrite your calculateDiscount function from earlier as a function expression assigned to a const variable instead of a declaration.

Arrow functions

A shorter, modern syntax for writing function expressions, introduced in ES6. Widely used for short functions, and behaves differently from regular functions with respect to this keyword (covered in Chapter 14).

Values / Details

Concept / Value	What it does
Full form	const add = (a, b) => { return a + b; };
Implicit return (single expression, no braces)	const add = (a, b) => a + b; automatically returns the expression's result without needing the return keyword.
Single parameter	Parentheses around a single parameter are optional: name => `Hi ${name}`.

Example

```
const calculateTotal = (price, quantity) => price * quantity;
const square = n => n * n;

console.log (calculateTotal(500, 3)); // 1500
console.log(square(4));               // 16
```

> **➖ Exercise**
> Rewrite your calculateTotal function from the first example in this chapter as a one-line arrow function using an implicit return.

Scope (global, function, block)

Determines where in your code a variable is accessible.

Values / Details

Concept / Value	What it does
Global scope	Variables declared outside any function or block are accessible everywhere in the file.
Function scope	Variables declared with var (or any variable, including parameters) inside a function are only accessible within that function.
Block scope	Variables declared with let or const inside a { } block (including if statements and loops) are only accessible within that specific block.

Example

```
const companyName = "Zolax Tech"; // global

function showPlan() {
  const plan = "Pro"; // function-scoped
  if (true) {
    const price = 1500; // block-scoped only exists inside this if
    console.log (plan, price);
  }
  // console.log(price); // would throw an error here
}
```

> **➖ Exercise**
> Write a short function containing an if block with its own let variable inside. Try to log that variable from outside the if block (in a comment, note what error you'd expect) to confirm your understanding of block scope.

Closures

A closure happens when a function 'remembers' the variables from the scope it was created in, even after that outer function has finished running. This is one of JavaScript's most powerful and initially confusing features, commonly used to create private state.

Example

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

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

> **➖ Exercise**
> Write a createQuoteCounter function similar to the example that returns an inner function tracking how many quote requests have been submitted. Call the returned function three times and log each result to confirm the count persists between calls.

Higher-order functions

A function that either accepts another function as an argument, or returns a function as its result (createCounter above is one example). This concept underlies many built-in array methods covered in Chapter 6.

Example

```
function processOrder(price, callback) {
  const total = price * 1.15; // add 15% tax
  callback(total);
}

processOrder(1000, (total) => {
  console.log(`Total with tax: ${total}`);
});
```

> **➖ Exercise**
> Write a processOrder function that accepts a price and a callback function, calculates a total with tax added, and passes that total into the callback to be logged.

Rest parameters

Let’s a function accept an unlimited number of arguments, automatically collected into a real array inside the function using the ...name syntax.

Example

```
function sumPrices(...prices) {
  return prices.reduce((total, price) => total + price, 0);
}

console.log(sumPrices(500, 1500, 300)); // 2300
```

> **➖ Exercise**
> Write a sumPrices function using rest parameters that can accept any number of price arguments and returns their total.

Chapter Summary Project

> **🏁 Chapter Project**
> Build a small 'quote builder' toolkit of functions for Zolax Tech: a calculateSubtotal(...prices) function using rest parameters, a calculateDiscount(subtotal, percent) arrow function, and a formatQuote(subtotal, discount) function that returns a template-literal summary string. Chain them together by calling each function with the previous one's result, and log the final formatted quote to the console.

Chapter 6: Arrays & Array Methods

Arrays store ordered lists of data client lists, service names, prices. This chapter covers creating, reading, and transforming them using JavaScript's powerful built-in methods.

Creating & accessing arrays

Arrays are written with square brackets; individual items are accessed by their zero-based index (the first item is index 0).

Values / Details

Concept / Value	What it does
.length	The number of items currently in the array.
array[array.length - 1]	A common pattern for accessing the LAST item, since there's no negative-indexing shortcut in plain JavaScript.

Example

```
const services = ["Web Development", "Mobile Apps", "UI/UX Design"];
console.log(services[0]);                    // "Web Development"
console.log(services.length);                  // 3
console.log(services[services.length - 1]);    // "UI/UX Design"
```

> ➖ **Exercise**
> Create an array of five client names and log the first, the last (using .length - 1), and the total count.

.push() & .pop()

Add or remove an item from the END of an array, modifying the original array directly (these are 'mutating' methods).

Values / Details

Concept / Value	What it does
.push(item)	Adds an item to the end and returns the new length.
.pop()	Removes and returns the last item.

Example

```
const queue = ["Client A", "Client B"];
queue.push("Client C");
console.log(queue); // ["Client A", "Client B", "Client C"]
const served = queue.pop();
console.log(served, queue); // "Client C" ["Client A", "Client B"]
```

> ➖ **Exercise**
> Simulate a support queue: push three client names into an array, then pop the last one off and log both the removed client and the remaining queue.

.shift() & .unshift()

Add or remove an item from the BEGINNING of an array useful for a 'first in, first out' queue, since pop()/push() work from the end.

Values / Details

Concept / Value	What it does
.unshift(item)	Adds an item to the start.
.shift()	Removes and returns the first item.

Example

```
const queue = ["Client B", "Client C"];
queue.unshift("Client A"); // add to front
console.log(queue); // ["Client A", "Client B", "Client C"]
const nextUp = queue.shift(); // remove from front
console.log(nextUp, queue); // "Client A" ["Client B", "Client C"]
```

> **➖ Exercise**
> Rebuild the support queue example using shift() and unshift() instead, to serve clients in the exact order they arrived (first-in, first-out).

.includes() & .indexOf()

Search an array for a specific value.

Values / Details

Concept / Value	What it does
.includes(value)	Returns true or false depending on whether the array contains that exact value.
.indexOf(value)	Returns the index of the first match, or -1 if not found.

Example

```
const services = ["Web Development", "Mobile Apps"];
console.log(services.includes("Mobile Apps")); // true
console.log(services.indexOf("SEO"));          // -1
```

> **➖ Exercise**
> Check whether your services array from earlier includes a specific service using .includes(), and find the index of one specific item using .indexOf().

.slice() & .splice()

Two similarly-named but very different methods for extracting or modifying portions of an array.

Values / Details

Concept / Value	What it does
.slice(start, end)	Returns a NEW array containing a copy of the selected range, WITHOUT modifying the original array.
.splice(start, deleteCount, ...items)	MODIFIES the original array directly removes deleteCount items starting at start, and can simultaneously insert new items in their place.

Example

```
const services = ["Web", "Mobile", "Design", "SEO", "Hosting"];
const firstTwo = services.slice(0, 2);
console.log(firstTwo); // ["Web", "Mobile”] original unchanged

services.splice(1, 1, "Mobile Apps"); // remove 1 item at index 1, insert a replacement
console.log(services); // ["Web", "Mobile Apps", "Design", "SEO", "Hosting"]
```

> **➖ Exercise**
> Use .slice() to extract just the first three items of your services array into a new array without changing the original, then use .splice() to replace one specific item in the original array with an updated name.

.forEach()

Runs a given function once for every item in the array the array-method equivalent of a for...of loop, but with a slightly different syntax and no built-in way to break out early.

Example

```
const services = ["Web Development", "Mobile Apps", "UI/UX Design"];

services.forEach((service, index) => {
  console.log(`${index + 1}. ${service}`);
});
```

> **➖ Exercise**
> Use .forEach() to log every item in your services array with a number prefix, using both the item and its index parameters.

.map()

Creates a BRAND NEW array by transforming every item using a given function, leaving the original array unchanged. One of the most frequently used array methods in real projects.

Example

```
const prices = [500, 1500, 3000];
const pricesWithTax = prices.map((price) => price * 1.15);
console.log(pricesWithTax); // [575, 1725, 3450]
console.log(prices);        // original unchanged: [500, 1500, 3000]
```

> **➖ Exercise**
> Use .map() to create a new array where every price in your prices array has a 15% tax added, and confirm the original array is unchanged afterward.

.filter()

Creates a new array containing only the items for which a given function returns true used to narrow a list down based on a condition.

Example

```
const leadScores = [30, 85, 55, 92, 15];
const hotLeads = leadScores.filter((score) => score >= 80);
console.log(hotLeads); // [85, 92]
```

> **➖ Exercise**
> Filter your prices array (or a new array of project budgets) down to only the values above a certain threshold, and log the filtered result.

.reduce()

Boils an entire array down to a single value (a total, an average, a combined object) by repeatedly applying a function that carries an 'accumulator' forward through each item one of the more advanced but powerful array methods.

Values / Details

Concept / Value	What it does
reduce((accumulator, currentItem) => { ... }, initialValue)	The initialValue sets the accumulator's starting point before the first item is processed.

Example

```
const prices = [500, 1500, 3000];
const total = prices.reduce((sum, price) => sum + price, 0);
console.log(total); // 5000
```

> **➖ Exercise**
> Use .reduce() to calculate the total of all prices in an array, starting the accumulator at 0, and compare the result to what you'd get by manually adding them.

.find() & .findIndex()

Search an array using a condition function rather than an exact value match.

Values / Details

Concept / Value	What it does
.find(fn)	Returns the FIRST item for which fn returns true, or undefined if none match.
.findIndex(fn)	Returns the index of the first match, or -1 if none match.

Example

```
const clients = [
  { name: "Selam Traders", plan: "Basic" },
  { name: "Habesha Foods", plan: "Pro" },
];
const proClient = clients.find((c) => c.plan === "Pro");
console.log(proClient.name); // "Habesha Foods"
```

> **➖ Exercise**
> Create an array of client objects like the example, and use .find() to locate the first client on the "Pro" plan, logging their name.

.sort()

Sorts an array's items IN PLACE (modifying the original). By default, it sorts as strings, which produces WRONG results for numbers unless you provide a compare function.

Values / Details

Concept / Value	What it does
Default (no arguments)	Converts items to strings and sorts alphabetically [10, 1, 2].sort() incorrectly gives [1, 10, 2].
(a, b) => a - b	A compare function that correctly sorts numbers in ascending order; (a, b) => b - a sorts descending.

Example

```
const prices = [1500, 500, 3000];
prices.sort((a, b) => a - b);
console.log(prices); // [500, 1500, 3000]
```

> **➖ Exercise**
> Sort an array of leadScores numbers in descending order (highest first) using .sort() with the correct compare function, and explain in a comment why the default .sort() alone would give the wrong result on numbers.

.join()

Combines every item of an array into a single string, with a chosen separator placed between each item.

Example

```
const services = ["Web", "Mobile", "Design"];
console.log(services.join(", ")); // "Web, Mobile, Design"
```

> **➖ Exercise**
> Use .join() to turn your services array into a single comma-separated sentence suitable for displaying on the page.

Chapter Summary Project

> **🏁 Chapter Project**
> Build a 'client dashboard' script: create an array of at least five client objects (each with name, plan, and monthlyValue), use. filter() to find only clients on the "Pro" plan, use .map() to create a list of just their names, use .reduce() to calculate total monthly revenue across ALL clients, and use .sort() to rank clients from highest to lowest monthlyValue. Log each result clearly labeled.

Chapter 7: Objects & Object Methods

Objects group related data and behavior together under one variable the backbone of representing real-world things like a client, a product, or a form's data in code.

Creating & accessing objects

Objects are written with curly braces containing key: value pairs. Keys can be accessed with dot notation or bracket notation.

Values / Details

Concept / Value	What it does
Dot notation	object.propertyName the most common and readable way, used when you know the exact property name in advance.
Bracket notation	object["propertyName"] required when the property name is stored in a variable, or contains spaces/special characters.

Example

```
const client = {
  name: "Selam Traders",
  plan: "Pro",
  monthlyValue: 1500,
};
```

```
console.log(client.name);       // dot notation
const key = "plan";
console.log(client[key]);       // bracket notation, using a variable
```

> **➼ Exercise**
> Create a client object with at least four properties. Access one property using dot notation and a different one using bracket notation with a variable holding the key name.

Adding, updating & deleting properties

Objects declared with const can still have their properties changed const only prevents reassigning the variable to a whole different object, not modifying what's inside it.

Values / Details

Concept / Value	What it does
delete object.property	Removes a property from an object entirely.

Example

```
const client = { name: "Selam Traders", plan: "Basic" };
client.plan = "Pro";          // update
client.email = "info@selam.com"; // add
delete client.email;          // delete
console.log(client); // { name: "Selam Traders", plan: "Pro" }
```

> **➼ Exercise**
> Take a client object, update its plan property, add a new phone property, and then delete that phone property, logging the object after each step.

Object methods (functions as properties)

A function stored as a property of an object is called a 'method', and can be called using dot notation. Inside a regular (non-arrow) method, this keyword refers to the object the method was called on.

Values / Details

Concept / Value	What it does
this	Refers to 'the object this method belongs to' lets a method access the object's own other properties without hard-coding the object's name.

Example

```
const client = {
  name: "Selam Traders",
  plan: "Pro",
  describe() {
    return `${this.name} is on the ${this.plan} plan.`;
  },
};

console.log(client.describe());
```

> **➖ Exercise**
> Add a describe() method to your client object that uses this to build and return a sentence combining several of the object's own properties.

Object.keys(), Object.values(), Object.entries()

Convert an object's contents into arrays so they can be looped over or processed with array methods like .map() and .forEach().

Values / Details

Concept / Value	What it does
Object.keys(obj)	Returns an array of just the property names.
Object.values(obj)	Returns an array of just the property values.
Object.entries(obj)	Returns an array of [key, value] pairs, useful for looping through both at once.

Example

```
const client = {name: "Selam Traders", plan: "Pro", monthlyValue: 1500};

console.log(Object.keys(client)); // ["name", "plan", "monthlyValue"]
console.log(Object.values(client)); // ["Selam Traders", "Pro", 1500]

Object.entries(client).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});
```

> **➖ Exercise**
> Use Object.entries() together with .forEach() to log every key and value pair from your client object in a readable 'key: value' format.

Object destructuring

A concise way to unpack an object's properties directly into standalone variables, avoiding repeated object.property access.

Values / Details

Concept / Value	What it does
const { a, b } = object;	Creates variables a and b matching the object's property names.
const { a: newName } = object;	Renames the extracted variable.
const { a = "default" } = object;	Provides a fallback if the property doesn't exist on the object.

Example

```
const client = {name: "Selam Traders", plan: "Pro", monthlyValue: 1500};
const {name, plan} = client;
console.log (name, plan); // "Selam Traders" "Pro"

function describeClient({ name, plan }) {
  return `${name} is on the ${plan} plan. `;
}
console.log(describeClient(client));
```

> **➖ Exercise**
> Destructure name and plan directly out of your client object into standalone variables, and rewrite one of your earlier functions to accept a client object and destructure its needed properties directly in the function's parameters.

Spread operator (objects)

Copies all of an existing object's properties into a brand new object using ...object, commonly used to create an updated copy without mutating the original especially important in frameworks like React.

Example

```
const client = {name: "Selam Traders", plan: "Basic" };
const upgradedClient = {...client, plan: "Pro”};

console.log(client);        // unchanged: plan is still "Basic"
console.log(upgradedClient); // { name: "Selam Traders", plan: "Pro" }
```

> **➖ Exercise**
> Use the spread operator to create an 'upgraded' copy of a client object with a different plan value, while confirming the original object remains completely unchanged.

Nested objects & arrays of objects

Objects can contain other objects or arrays as property values, modeling more complex real-world data like a client that has an array of past projects, each of which is its own object.

Example

```
const client = {
  name: "Selam Traders",
  contact: {
    email: "info@selam.com",
    phone: "+251900000000",
  },
  projects: [
    { title: "Website Redesign", status: "Complete" },
    { title: "Mobile App", status: "In Progress" },
  ],
};

console.log(client.contact.email);
console.log(client.projects[1].title);
```

> **✏ Exercise**
> Build a nested client object with a contact sub-object and a projects array of project objects, then access one deeply nested value (like a project's status) directly by chaining property/index access.

Chapter Summary Project

> **🏁 Chapter Project**
> Build a complete 'client record' object for Zolax Tech with nested contact info and an array of project objects, add a describe() method using this, use Object.entries() to log all top-level properties in a loop, use destructuring to pull out just the name and contact, and use the spread operator to create an updated copy of the client with one property changed confirming the original stays untouched.

Chapter 8: The DOM Selecting & Manipulating Elements

The Document Object Model (DOM) is the browser's live representation of your HTML page as a tree of objects that JavaScript can read and change. This is where JavaScript starts actually affecting what visitors see on zolaxtech.com.et.

document.querySelector()

Returns the FIRST element in the page matching a given CSS selector (the same kind of selector you write in styles.css), or null if nothing matches the most commonly used way to grab a single element.

Example

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

```
const firstCard = document.querySelector(".card");
const logo = document.querySelector("#site-logo");
console.log(heading, firstCard, logo);
```

> **➼ Exercise**
> Use querySelector to select your page's main <h1>, one element by class, and one element by id, logging all three to confirm they're found correctly.

document.querySelectorAll()

Returns EVERY element matching a given CSS selector as a NodeList (an array-like collection), which can be looped over with. forEach().

Example

```
const cards = document.querySelectorAll(".card");
console.log(cards.length);

cards.forEach((card) => {
  console.log(card);
});
```

> **➼ Exercise**
> Select every element on your page sharing your .card class using querySelectorAll, log how many were found, and loop through them with .forEach().

getElementById / getElementsByClassName (legacy)

Older, more specific selection methods that predate querySelector. Still seen in older code, but querySelector/querySelectorAll are generally preferred today since they accept any CSS selector, not just an id or class.

Example

```
const logo = document.getElementById("site-logo");
const cards = document.getElementsByClassName("card"); // returns a live HTMLCollection, not an array
```

> **➼ Exercise**
> Select the same element two different ways once with getElementById and once with querySelector('#id') and confirm in the console that both return the exact same element.

.textContent & .innerHTML

Read or change what's displayed inside an element.

Values / Details

Concept / Value	What it does
.textContent	Gets or sets an element's text ONLY, treating any markup as plain text the safer choice when inserting untrusted or dynamic text.
.innerHTML	Gets or sets an element's content AS HTML, meaning any tags you assign to it will actually render powerful, but inserting untrusted user input this way is a security risk (XSS), so use .textContent whenever you're just inserting plain text.

Example

```
const heading = document.querySelector("h1");
heading.textContent = "Welcome to Zolax Tech!";

const promo = document.querySelector(".promo");
promo.innerHTML = "<strong>Limited-time offer!</strong>";
```

> **➖ Exercise**
> Select your homepage's main heading and change its text using .textContent, then select a separate element and insert a small piece of bold HTML into it using .innerHTML.

.setAttribute() / .getAttribute() / .removeAttribute()

Read, set, or remove any HTML attribute on an element directly from JavaScript.

Example

```
const link = document.querySelector("a.cta");
link.setAttribute("href", "contact.html");
console.log(link.getAttribute("href"));
link.removeAttribute("target");
```

> **➖ Exercise**
> Select a link on your page and use setAttribute to change its href, then use getAttribute to log the new value back out to confirm the change.

.classList

The modern way to add, remove, toggle, or check CSS classes on an element from JavaScript works together with your existing CSS rather than setting inline styles directly.

Values / Details

Concept / Value	What it does
.add('class')	Adds a class.
.remove('class')	Removes a class.
.toggle('class')	Adds the class if it's missing, removes it if it's present perfect for things like a mobile menu open/close button.
.contains('class')	Returns true or false depending on whether the class is currently present.

Example

```
const card = document.querySelector(".card");
card.classList.add("featured");
card.classList.toggle("active");
console.log(card.classList.contains("featured")); // true
```

> **➖ Exercise**
> Select one of your service cards and use classList.add() to give it a 'featured' class you've already styled in CSS, then use classList.toggle() on a separate element to switch a class on and off, confirming the visual change in the browser.

.style (inline styles from JS)

Directly sets an individual CSS property on an element, similar to writing an inline style attribute useful for values that must be calculated dynamically, but classList is preferred for anything that could instead be a predefined CSS class.

Example

```
const banner = document.querySelector(".banner");
banner.style.backgroundColor = "crimson";
banner.style.padding = "20px";
```

> **➖ Exercise**
> Select an element and change one CSS property directly through .style, then consider (in a comment) whether that same change could instead be done more cleanly using classList.add() with a pre-written CSS class.

Creating & inserting elements

JavaScript can build entirely new HTML elements and insert them into the page the foundation of dynamically generated content like a list of search results or client testimonials loaded from data.

Values / Details

Concept / Value	What it does
document.createElement('tag')	Creates a new, empty element in memory (not yet visible on the page).
parent.appendChild(el)	Inserts the new element as the last child of a chosen parent.
parent.append(el)	A more flexible modern alternative to appendChild that can also accept plain text.
el.remove()	Removes an element from the page entirely.

Example

```
const list = document.querySelector("#service-list");
const newItem = document.createElement("li");
newItem.textContent = "SEO Optimization";
list.appendChild(newItem);
```

> **➖ Exercise**
> Create a new <li> element, set its text to a new service name, and append it to an existing <ul> on your page, confirming it appears in the browser without editing the HTML file directly.

Traversing the DOM

Navigate between related elements without needing a new query for each one.

Values / Details

Concept / Value	What it does
.parentElement	The element's direct parent.
.children	A live collection of the element's direct child elements.
.nextElementSibling / .previousElementSibling	The next/previous element at the same level.
.closest('selector')	Walks UP the tree from the element and returns the nearest ancestor (or the element itself) matching a given selector very useful inside event handlers.

Example

```
const card = document.querySelector(".card");
console.log(card.parentElement);
console.log(card.nextElementSibling);
const section = card.closest("section");
console.log(section);
```

➖ Exercise

Select one of your cards and log its parentElement, its nextElementSibling (if it has one), and use .closest() to find its nearest enclosing <section>.

Chapter Summary Project

🏁 Chapter Project

Write a script that: selects every .card on your page and adds a 'loaded' class to each one using classList, dynamically creates and appends a brand-new service card (built entirely with createElement and textContent/classList, not copy-pasted HTML) to your services section, and changes your page's main heading text to include the current number of services found via querySelectorAll(...).length.

Chapter 9: Events & Event Handling

Events are how JavaScript responds to what a visitor DOES on the page clicking, typing, scrolling, submitting a form. This is what makes zolaxtech.com.et feel interactive rather than static.

addEventListener()

Attaches a function (the 'event handler' or 'listener') to an element, which runs automatically whenever a specified event occurs on that element. This is the standard, modern way to handle events preferred over inline HTML attributes like onclick.

Values / Details

Concept / Value	What it does
element.addEventListener('event', handlerFunction)	The event name is given as a string without 'on' (e.g. 'click', not 'onclick').
removeEventListener('event', handlerFunction)	Detaches a previously attached listener (requires a reference to the same named function, not an anonymous one).

Example

```
const quoteButton = document.querySelector("#get-quote");

quoteButton.addEventListener("click", function () {
  console.log("Quote button was clicked!");
});
```

➖ Exercise

Select a button on your page and add a click event listener that logs a message to the console confirming it was clicked.

The event object

Every event handler function automatically receives an 'event object' as its first argument, containing details about exactly what happened which key was pressed, which element was clicked, the mouse coordinates, and more.

Values / Details

Concept / Value	What it does
event.target	The exact element the event actually occurred on essential for event delegation (see below).
event.type	The name of the event that fired, e.g. "click".
event.preventDefault()	Stops the browser's default behavior for that event most commonly used to prevent a form from actually submitting/reloading the page, or a link from navigating.

Example

```
const link = document.querySelector("a.disabled-link");

link.addEventListener("click", function (event) {
  event.preventDefault(); // stops the browser from following the link
  console.log("Clicked:", event.target.textContent);
});
```

> **➖ Exercise**
> Add a click listener to a link that calls event.preventDefault() to stop it from navigating, and logs event.target to confirm exactly which element was clicked.

Click, mouse & keyboard events

Beyond click, many other event types let you respond to different kinds of interaction.

Values / Details

Concept / Value	What it does
click / dblclick	A single or double mouse click.
mouseenter / mouseleave	The pointer entering/leaving an element (doesn't 'bubble' to children, unlike mouseover/mouseout).
keydown / keyup	A keyboard key being pressed down or released, giving access to event.key for which key was involved.
submit	A <form> being submitted almost always paired with event.preventDefault() so you can validate before allowing (or replacing) the actual submission.

Concept / Value	What it does
input	Fires every time a text field's value changes, immediately as the user types ideal for live character counters or search-as-you-type.
change	Fires when a field's value changes AND it loses focus (or, for checkboxes/selects, immediately on selection) better suited than input for things like a dropdown selection.

Example

```
const searchBox = document.querySelector("#search");

searchBox.addEventListener("input", function (event) {
  console.log("Current value:", event.target.value);
});

document.addEventListener("keydown", function (event) {
  if (event.key === "Escape") {
    console.log("Escape key pressed  closing any open menu");
  }
});
```

> **➖ Exercise**
> Add an input event listener to a text field that logs its current value on every keystroke, and add a document-level keydown listener that logs a message specifically when the Escape key is pressed.

Event delegation

Instead of attaching a separate listener to every individual item (which is wasteful and misses items added later), you attach ONE listener to a shared PARENT element, and use event.target inside it to figure out which specific child was actually interacted with. This is a core, widely-used pattern in real applications.

Example

```
const cardContainer = document.querySelector("#services");

cardContainer.addEventListener("click", function (event) {
  const card = event.target.closest(".card");
  if (!card) return; // click wasn't on or inside a card
  console.log("Clicked card:", card.querySelector("h3").textContent);
});
```

> **➖ Exercise**
> Attach a single click listener to the container holding all your service cards (instead of one listener per card), and use event.target.closest('.card') inside it to detect and log which specific card was clicked confirming it works even for cards added dynamically after the page loaded.

Form submission handling

Combines preventDefault() with reading form field values to handle a form entirely with JavaScript the foundation of client-side validation, covered fully in the next chapter.

Example

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

form.addEventListener("submit", function (event) {
  event.preventDefault();
  const nameInput = document.querySelector("#name");
  console.log("Form submitted by:", nameInput.value);
});
```

> **➖ Exercise**
>
> Add a submit listener to your quote/contact form that prevents the default page reload and logs the current value of the name field to the console instead.

Toggling visibility with events (practical pattern)

A very common combination: listening for a click on a button, then using classList.toggle() to show or hide another element the basis of mobile menus, FAQ accordions, and modals built without <details>.

Example

```
const menuButton = document.querySelector("#menu-toggle");
const nav = document.querySelector("nav");

menuButton.addEventListener("click", function () {
  nav.classList.toggle("open");
});
```

> **➖ Exercise**
>
> Build a working mobile menu toggle: a button that, when clicked, toggles an 'open' class on your <nav> element, paired with a CSS rule (from your CSS guide) that shows/hides the nav based on that class.

Chapter Summary Project

> **🏁 Chapter Project**
>
> Make your zolaxtech.com.et homepage interactive: wire up a working mobile menu toggle button using classList.toggle(), add a submit handler to your contact form that calls preventDefault() and logs every field's value, and use event delegation with a single listener on your card container to log which service card was clicked, including a highlight class added to the clicked card via classList.

Chapter 10: Forms & Validation with JavaScript

While HTML5's built-in validation (required, pattern, type="email") catches many basic cases, JavaScript lets you build custom validation logic, live feedback, and full control over what happens when a visitor submits the Zolax Tech contact form.

Reading form field values

Every form input's current value is available through its .value property (checkboxes and radios use. checked instead).

Values / Details

Concept / Value	What it does
input.value	The current text/number/etc. in a text, email, number, textarea, or select field.
checkbox.checked	true or false depending on whether a checkbox or radio button is currently selected.

Example

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

console.log(nameInput.value);
console.log(newsletterCheckbox.checked);
```

> **➖ Exercise**
> Select the name field and a checkbox from your quote form, and log both their current .value and .checked properties to the console.

Custom validation logic

Beyond simply checking whether a field is empty, you can write any logic you want checking a minimum length, a valid format, or a business rule specific to Zolax Tech.

Example

```
function isValidEmail(email) {
  return email.includes("@") && email.includes(".");
}

function validateForm(name, email, budget) {
  const errors = [];
  if (name.trim() === "") errors.push("Name is required.");
  if (!isValidEmail(email)) errors.push("A valid email is required.");
  if (budget < 500) errors.push("Minimum project budget is 500 ETB.");
  return errors;
}
```

```
console.log(validateForm("", "not-an-email", 200));
```

> **➖ Exercise**
> Write your own validateForm function that checks a name, email, and budget using the logic shown, returning an array of error message strings, and test it by calling it with both invalid and fully valid sample data.

Displaying error messages

Take the array of validation errors and actually show it to the visitor by creating or updating an element on the page rather than only logging to a console the visitor will never see.

Example

```
function showErrors(errors) {
  const errorBox = document.querySelector("#form-errors");
  errorBox.innerHTML = "";
  errors.forEach((message) => {
    const li = document.createElement("li");
    li.textContent = message;
    errorBox.appendChild(li);
  });
}
```

> **➖ Exercise**
> Add an empty <ul id="form-errors"> to your contact.html, then write a showErrors function that clears it and appends a new <li> for each validation error message, and test it with a sample array of error strings.

Full form-submission validation flow

Combines everything so far: intercept the submit event, read the values, validate them, and either show errors or proceed (e.g. show a success message) entirely without a page reload.

Example

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

form.addEventListener("submit", function (event) {
  event.preventDefault();

  const name = document.querySelector("#name").value;
  const email = document.querySelector("#email").value;
  const budget = Number(document.querySelector("#budget").value);
```

```
  const errors = validateForm(name, email, budget);

  if (errors.length > 0) {
    showErrors(errors);
  } else {
    document.querySelector("#form-errors").innerHTML = "";
    console.log("Form is valid  ready to submit:", { name, email, budget });
  }
});
```

> **➖ Exercise**
> Wire your validateForm and showErrors functions together inside a single submit event listener on your real contact form, so submitting with invalid data shows visible errors on the page, and submitting with valid data clears any old errors and logs a success message.

Live/inline validation on blur

Rather than waiting for the whole form to be submitted, you can validate a SINGLE field as soon as the visitor leaves it (the blur event), giving much faster feedback.

Values / Details

Concept / Value	What it does
blur event	Fires when an element loses focus the visitor has clicked or tabbed away from it.

Example

```
const emailInput = document.querySelector("#email");

emailInput.addEventListener("blur", function () {
  if (!isValidEmail(emailInput.value)) {
    emailInput.classList.add("invalid");
  } else {
    emailInput.classList.remove("invalid");
  }
});
```

> **➖ Exercise**
> Add a blur event listener to your email field that adds an 'invalid' class (which you can style with a red border in CSS) if the value fails your isValidEmail check, and removes it once the value becomes valid.

Resetting a form with JavaScript

Clears every field in a form back to its default values programmatically useful after a successful submission.

Values / Details

Concept / Value	What it does
form.reset()	Resets all fields in the given form element, equivalent to clicking a type="reset" button.

Example

```
form.reset();
document.querySelector("#form-errors").innerHTML = "";
```

➖ Exercise

After logging a successful validation in your submit handler, call form.reset() to clear the form automatically, simulating what a real site would do after successfully sending the data to a server.

Chapter Summary Project

🏁 Chapter Project

Build complete custom JavaScript validation for your Zolax Tech quote form: a validateForm function checking name, email format, and a minimum budget; a showErrors function that displays messages in a visible list on the page; a submit handler that ties them together and calls form.reset() on success; and at least one blur-based inline validation on a single field for instant feedback.

Chapter 11: Asynchronous JavaScript

Some operations fetching data from a server, waiting for a timer take time to complete. This chapter covers how JavaScript handles that without freezing the entire page while it waits.

The problem: synchronous vs asynchronous

By default, JavaScript runs one line at a time, in order (synchronously), and each line must finish before the next one starts. Some tasks like a network request would freeze the whole page for seconds if handled this way, so JavaScript instead lets them run in the background and notifies your code when they're done (asynchronously).

Example

```
console.log("1: Starting quote request");
setTimeout(() => {
  console.log("3: Quote data received (after a delay)");
}, 1000);
console.log("2: Continuing with other work while we wait");
```

```
// Console output order: 1, 2, then 3 (after ~1 second)
```

> **➼ Exercise**
> Run the example exactly as written and predict the ORDER the three console.log messages will appear before running it then confirm your prediction was correct.

setTimeout() & setInterval()

Schedule a function to run after a delay (once, or repeatedly).

Values / Details

Concept / Value	What it does
setTimeout(fn, delayMs)	Runs fn a single time, after at least delayMs milliseconds have passed.
setInterval(fn, delayMs)	Runs fn repeatedly, every delayMs milliseconds, until stopped.
clearInterval(intervalId)	Stops a running setInterval, using the ID it returned when created.

Example

```
const timeoutId = setTimeout(() => {
  console.log("Showing a 'Need help?' popup after 5 seconds");
}, 5000);

let seconds = 0;
const intervalId = setInterval(() => {
  seconds++;
  console.log(`${seconds} seconds on page`);
  if (seconds === 5) clearInterval(intervalId);
}, 1000);
```

> **➼ Exercise**
> Use setTimeout to log a 'Need help? Contact Zolax Tech' message after 3 seconds, and use setInterval combined with clearInterval to log a counter once per second that automatically stops after 5 counts.

Callbacks

The original way to handle asynchronous results: pass a function to be called later, once the async operation finishes. Deeply nested callbacks (waiting on one async result inside another) create hard-to-read 'callback hell', which Promises were designed to solve.

Example

```
function fetchClientData(clientId, callback) {
  setTimeout(() => {
    callback({ id: clientId, name: "Selam Traders" });
  }, 1000);
}

fetchClientData(1, function (client) {
  console.log("Received:", client.name);
});
```

> **➥ Exercise**
> Write a fetchClientData function using setTimeout and a callback, similar to the example, and call it with a callback that logs the received client's name.

Promises

An object representing a value that will be available LATER either successfully (resolved) or with an error (rejected). Promises make chains of async operations far more readable than nested callbacks.

Values / Details

Concept / Value	What it does
new Promise((resolve, reject) => { ... })	Creates a promise; call resolve(value) on success or reject(error) on failure inside the executor function.
.then(callback)	Runs when the promise resolves successfully, receiving the resolved value.
.catch(callback)	Runs if the promise rejects, receiving the error.
.finally(callback)	Runs regardless of success or failure useful for cleanup like hiding a loading spinner.

Example

```
function fetchClientData(clientId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (clientId > 0) {
        resolve({ id: clientId, name: "Selam Traders" });
      } else {
        reject(new Error("Invalid client ID"));
      }
    }, 1000);
  });
}
```

```
fetchClientData(1)
  .then((client) => console.log("Received:", client.name))
  .catch((error) => console.error("Error:", error.message))
  .finally(() => console.log("Request finished"));
```

> **➖ Exercise**
> Rewrite your fetchClientData function to return a Promise instead of using a callback, resolving with client data on a valid ID and rejecting with an Error on an invalid one (like 0 or a negative number). Chain .then(), .catch(), and .finally() onto a call to it.

async / await

Modern, cleaner syntax built on top of Promises that lets asynchronous code READ almost like ordinary synchronous code, avoiding long. then () chains.

Values / Details

Concept / Value	What it does
async function	Marks a function as asynchronous, meaning it automatically returns a Promise and can use await inside it.
await promise	Pauses execution of the async function (without freezing the rest of the page) until the promise settles, then returns its resolved value directly.
try / catch (with await)	The standard way to handle rejected promises when using await, since a rejected awaited promise throws an error.

Example

```
async function loadClient(clientId) {
  try {
    const client = await fetchClientData(clientId);
    console.log("Received:", client.name);
  } catch (error) {
    console.error("Error:", error.message);
  } finally {
    console.log("Request finished");
  }
}

loadClient(1);
```

➼ Exercise

Rewrite your .then()/.catch() chain from the previous exercise as an async function using await inside a try/catch/finally block instead, and confirm it behaves identically for both a valid and an invalid client ID.

The Fetch API

The browser's built-in way to make real network requests to a server or public API, returning a Promise this is how a real version of zolaxtech.com.et would load live data (like blog posts or pricing) without a page reload.

Values / Details

Concept / Value	What it does
fetch(url)	Starts a GET request and returns a Promise that resolves with a Response object once the server responds (NOT once the body is fully read).
response.json()	Reads and parses the response body as JSON, itself returning another Promise.
response.ok	A boolean indicating whether the server responded with a success status code (200-299) fetch does NOT reject automatically on 404/500 errors, so this must be checked manually.

Example

```
async function loadPosts() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/posts?_limit=3");
    if (!response.ok) throw new Error(`Server responded with ${response.status}`);
    const posts = await response.json();
    posts.forEach((post) => console.log(post.title));
  } catch (error) {
    console.error("Failed to load posts:", error.message);
  }
}

loadPosts();
```

➼ Exercise

Use fetch with async/await to request data from https://jsonplaceholder.typicode.com/posts?_limit=3 (a free public testing API), check response.ok, parse the JSON, and log each post's title to the console inside a try/catch block.

Using fetched data to update the page

Combines Chapter 8's DOM skills with fetch: instead of only logging fetched data to the console, insert it into the actual page for the visitor to see.

Example

```
async function renderPosts() {
  const list = document.querySelector("#news-list");
  const response = await fetch("https://jsonplaceholder.typicode.com/posts?_limit=3");
  const posts = await response.json();

  posts.forEach((post) => {
    const li = document.createElement("li");
    li.textContent = post.title;
    list.appendChild(li);
  });
}

renderPosts();
```

➖ Exercise

Add an empty <ul id="news-list"> to a page, then write a renderPosts function that fetches sample posts and appends each one as a new <li>, so the fetched data actually appears visibly on your page rather than only in the console.

Chapter Summary Project

🏁 Chapter Project

Build a small 'Latest Updates' feature for zolaxtech.com.et: write an async function that fetches sample data from https://jsonplaceholder.typicode.com/posts?_limit=5, handles both success and failure with try/catch, and renders each item into a list on the page using DOM methods from Chapter 8. Add a loading message that appears before the fetch starts and is removed once the data has rendered (or an error message is shown, if the request fails).

Chapter 12: Modern ES6+ Features

This chapter gathers the modern JavaScript syntax that has become standard in professional code since ES6 (2015) much of it already used in earlier chapters, now explained together as a toolkit.

Template literals (recap)

Backtick strings supporting embedded expressions and real multi-line text, covered in Chapter 3 foundational enough to be used throughout every chapter since.

Example

```
const name = "Selam";
const message = `Hello, ${name}!
Welcome to Zolax Tech.`;
console.log(message);
```

> **➖ Exercise**
> Write a multi-line template literal (spanning at least two lines directly in the backticks) summarizing a client's order, embedding at least two variables.

Destructuring (arrays)

Similar to object destructuring (Chapter 7), but unpacks array items by POSITION into individual variables.

Values / Details

Concept / Value	What it does
const [a, b] = array;	Assigns the first two items to a and b.
const [first, , third] = array;	Skips an item using an empty slot.
Swapping values	[a, b] = [b, a]; swaps two variables' values in a single line without a temporary variable.

Example

```
const topServices = ["Web Development", "Mobile Apps", "UI/UX Design"];
const [primary, secondary] = topServices;
console.log(primary, secondary); // "Web Development" "Mobile Apps"

let a = 1, b = 2;
[a, b] = [b, a];
console.log(a, b); // 2 1
```

> **➖ Exercise**
> Destructure the first two items of an array into named variables, and use array destructuring to swap the values of two existing variables in a single line.

Spread operator (arrays)

Expands an array's items in place used to copy an array, merge multiple arrays, or pass an array's items as individual function arguments.

Example

```
const coreServices = ["Web", "Mobile"];
const allServices = [...coreServices, "Design", "SEO"];
console.log(allServices); // ["Web", "Mobile", "Design", "SEO"]

const copy = [...coreServices]; // a real, independent copy
```

> **➖ Exercise**
> Use the spread operator to merge two separate arrays of services into one combined array, and separately create an independent copy of an array, confirming that changing the copy doesn't affect the original.

Optional chaining (?)

Safely accesses a deeply nested property without throwing an error if an intermediate piece is missing returns undefined instead of crashing the script.

Example

```
const client = {name: "Selam Traders”}; // no 'contact' property

// Without optional chaining, this would throw an error:
console.log(client.contact?.email); // undefined, no crash
```

> **➖ Exercise**
> Create an object missing a nested property on purpose, and use optional chaining to safely access a deeply nested value without your script crashing, logging the resulting undefined.

Nullish coalescing (recap) & logical assignment

Provide clean shorthand for common 'use this value, or fall back to a default' patterns.

Values / Details

Concept / Value	What it does
??	Returns the right side only when the left side is null or undefined (Chapter 3 recap).
??=	Assigns the right-hand value ONLY if the variable is currently null or undefined.

Example

```
let discount = null;
discount ??= 0; // assigns 0, since discount was null
console.log(discount); // 0
```

> **➼ Exercise**
> Declare a variable set to null, use ??= to give it a default value, and log the result to confirm it only assigns when the original value was null/undefined.

Classes

A modern, more readable syntax for creating multiple similar objects (like many client records) that share the same structure and methods, built on top of JavaScript's underlying prototype system.

Values / Details

Concept / Value	What it does
constructor(params) { }	A special method that runs automatically when a new object is created with new, used to set up initial properties.
this.property = value;	Assigns a property to the specific instance being created.
methodName() { }	Regular methods, shared automatically by every instance of the class.
new ClassName(...)	Creates a new object ('instance') based on the class.

Example

```
class Client {
  constructor(name, plan) {
    this.name = name;
    this.plan = plan;
  }

  describe() {
    return `${this.name} is on the ${this.plan} plan.`;
  }
}

const client1 = new Client("Selam Traders", "Pro");
console.log(client1.describe());
```

> **➼ Exercise**
> Write a Client class with a constructor accepting name and plan, and a describe () method. Create two different instances using new and call describe() on each.

Class inheritance (extends)

Let’s one class build on another, inheriting its properties and methods while adding or overriding its own avoids duplicating shared logic across similar classes.

Values / Details

Concept / Value	What it does
extends ParentClass	Marks a class as inheriting from another.
super(...)	Calls the parent class's constructor, required before using this in a subclass's own constructor.

Example

```
class EnterpriseClient extends Client {
  constructor(name, accountManager) {
    super(name, "Enterprise");
    this.accountManager = accountManager;
  }

  describe() {
    return `${super.describe()} Managed by ${this.accountManager}.`;
  }
}

const bigClient = new EnterpriseClient("Habesha Foods", "Amanuel");
console.log(bigClient.describe());
```

> **➖ Exercise**
> Create an EnterpriseClient class that extends your Client class, adding an accountManager property and overriding describe() to include it while still reusing the parent's version via super.describe().

Modules (import / export)

Let you split JavaScript across multiple files and share code between them explicitly, instead of relying on one giant script.js or global variables the standard structure for any real project.

Values / Details

Concept / Value	What it does
export	Marks a variable, function, or class as available to other files, e.g. export function calculateTotal() { }.
export default	Marks a single 'main' export per file.
import { name } from './file.js';	Imports specific named exports from another file.
<script type="module">	Required on the HTML <script> tag for import/export to work directly in the browser.

Example

```
// pricing.js
export function calculateTotal(price, quantity) {
```

```
  return price * quantity;
}

// script.js
import { calculateTotal } from "./pricing.js";
console.log(calculateTotal(500, 3));

<!-- in HTML -->
<script type="module" src="script.js"></script>
```

➖ Exercise

Create a new file pricing.js exporting one function, import it into script.js, update your HTML's <script> tag to type="module", and confirm the imported function still works correctly when called.

Chapter Summary Project

🏁 Chapter Project

Refactor your Zolax Tech scripts using modern ES6+ syntax throughout: convert one repeated object shape (like your client records) into a Client class with a describe() method, create at least one subclass using extends and super, split your pricing-calculation functions into a separate pricing.js module and import them properly, and use optional chaining somewhere your data might realistically be incomplete.

Chapter 13: Error Handling & Debugging

Real code encounters unexpected situations a missing field, a failed network request, invalid input. This chapter covers catching those situations gracefully and finding bugs efficiently.

try / catch / finally

Let's you attempt code that might fail, and handle the failure gracefully instead of letting the whole script crash and stop running.

Values / Details

Concept / Value	What it does
try { }	Code that might throw an error.
catch (error) { }	Runs only if the try block throws; receives the error object.
finally { }	Runs regardless of whether an error occurred for cleanup that must always happen.

Example

```
try {
  const data = JSON.parse(“{invalid json ");
  console.log(data);
} catch (error) {
  console.error("Failed to parse data:", error.message);
} finally {
  console.log ("Parsing attempt finished.");
}
```

➼ Exercise

Write a try/catch block that attempts to JSON.parse a deliberately broken string, logs a friendly error message in the catch block instead of letting the script crash, and logs a 'finished' message in finally regardless of the outcome.

throw

Manually creates and raises an error, immediately stopping execution of the current function unless caught by an enclosing try/catch used to enforce your own business rules.

Values / Details

Concept / Value	What it does
throw new Error('message')	The standard way to raise an error with a descriptive message.

Example

```
function calculateDiscount(price, percent) {
  if (percent < 0 || percent > 100) {
    throw new Error ("Discount percent must be between 0 and 100.");
  }
  return price - price * (percent / 100);
}

try {
  calculateDiscount(1000, 150);
} catch (error) {
  console.error(error.message);
}
```

➼ Exercise

Add a throw statement to one of your existing functions from earlier chapters (like a discount or validation function) that enforces a business rule, and call it inside a try/catch to confirm the error is caught and logged cleanly.

Custom error handling for fetch/async code

Recap and extension of Chapter 11: real async code should always account for network failures, not just successful responses.

Example

```
async function loadClient(id) {
  try {
    const response = await fetch(`https://api.example.com/clients/${id}`);
    if (! response.ok) {
      throw new Error (`Client not found (status ${response.status})`);
    }
    return await response.json();
  } catch (error) {
    console.error("Could not load client:", error.message);
    return null;
  }
}
```

> **➖ Exercise**
>
> Write an async function using fetch that checks response.ok and throws a custom error message if the request failed, catches that error, logs it, and returns null so the rest of the program can keep running safely.

console methods for debugging

console.log is only one of several console methods, each useful for different debugging situations.

Values / Details

Concept / Value	What it does
console.error()	Logs in red, styled as an error useful for making real problems stand out from regular logs.
console.warn()	Logs in yellow, styled as a warning.
console.table()	Displays an array of objects as a readable table, extremely useful when inspecting lists of records like clients.
console.group() / console.groupEnd()	Visually nests a group of related log messages together, collapsible in DevTools.

Example

```
const clients = [
  {name: "Selam Traders", plan: "Pro”},
  {name: "Habesha Foods", plan: "Basic”},
];
console.table(clients);
```

```
console.group("Form Validation");
console.warn("Email field is empty");
console.error("Budget is below minimum");
console.groupEnd();
```

> **➖ Exercise**
> Use console.table() to display your array of client objects from Chapter 6 or 7 as a table, and use console.group()/console.groupEnd() to nest a console.warn() and a console.error() message together under one labeled group.

Using breakpoints in DevTools

Rather than sprinkling console.log () everywhere, you can pause your script's execution at an exact line directly in the browser and inspect every variable's current value.

Example

```
// No special code needed set a breakpoint directly
// on a line inside DevTools' Sources tab, for example:
function calculateTotal(price, quantity) {
  const total = price * quantity; // click the line number here
  return total;
}
```

> **➖ Exercise**
> Open DevTools' Sources tab, set a breakpoint inside one of your functions, trigger it from the page, and confirm you can see the function's parameter values while execution is paused. Step through one line at a time before letting it resume.

Common error types

Recognizing the built-in error types JavaScript throws helps you diagnose problems faster.

Values / Details

Concept / Value	What it does
SyntaxError	The code itself is written incorrectly and can't even be parsed, e.g. a missing closing bracket.
ReferenceError	You tried to use a variable that hasn't been declared, or isn't accessible in the current scope.
TypeError	You tried to do something invalid for a value's actual type, e.g. calling .toUpperCase() on a number, or reading a property of undefined.

Example

```
// TypeError example:
const client = undefined;
try {
  console.log(client.name);
} catch (error) {
  console.log(error.name, "-", error.message);
  // "TypeError - Cannot read properties of undefined (reading 'name')"
}
```

> **➖ Exercise**
> Deliberately write three small snippets that each trigger a different error type (SyntaxError, ReferenceError, TypeError you can just describe the SyntaxError one in a comment since it would stop the file from running), and use try/catch to catch and log the two you can safely run.

Chapter Summary Project

> **🏁 Chapter Project**
> Add production-quality error handling to your Zolax Tech quote form and fetch-based news feature from earlier chapters: wrap all form processing and async fetch logic in try/catch blocks with clear thrown error messages, log any errors using console.error() with a console.group() label, and set at least one breakpoint in DevTools while testing to step through your validation logic line by line.

Chapter 14: Advanced Topics & Best Practices

This final chapter ties together deeper mechanics of the language and the habits that separate working code from professional, maintainable code critical as your zolaxtech.com.et scripts grow.

Understanding this (deep dive)

The value of this depends entirely on HOW a function is called, not where it was defined a common source of confusion.

Values / Details

Concept / Value	What it does
Regular function as a method	this refers to the object the method was called on, e.g. client.describe() → this is client.
Regular function called alone	this is undefined in strict mode (or the global object otherwise) a common bug source when a method is passed as a callback and loses its object context.
Arrow functions	Do NOT have their own this they inherit this from the surrounding (enclosing) scope at the time they were defined, which makes them ideal for callbacks inside methods.

Example

```
const client = {
  name: "Selam Traders",
  services: ["Web", "Mobile"],
  logServices: function () {
    // arrow function here correctly inherits 'this' from logServices
    this.services.forEach((service) => {
      console.log(`${this.name}: ${service}`);
    });
  },
};

client.logServices();
```

> **➖ Exercise**
> Write an object with a method that loops over one of its own array properties using .forEach() with an arrow function callback, using this.propertyName inside that callback, and confirm it correctly refers back to the object.

The prototype chain (how classes really work)

Under the hood, JavaScript objects can be linked to another object (their 'prototype'), and when you access a property that doesn't exist directly on an object, JavaScript automatically looks up the chain. The class syntax from Chapter 12 is convenient syntax built on top of this same underlying mechanism.

Example

```
class Client {
  describe () {
    return `${this.name}'s plan: ${this.plan}`;
  }
}

const client1 = new Client ();
client1.name = "Selam Traders";
client1.plan = "Pro";

console.log (client1.describe()); // found via the prototype chain
console.log(Object.getPrototypeOf(client1) === Client.prototype); // true
```

> **➖ Exercise**
> Create a simple class with one method, instantiate it, and use Object.getPrototypeOf() to confirm in the console that the instance's prototype is indeed the class's own .prototype object.

Immediately Invoked Function Expressions (IIFE)

A function that runs immediately once it's defined, without needing to be called separately elsewhere historically used to avoid polluting the global scope before modules (Chapter 12) became standard. Still occasionally seen in older code or specific patterns.

Example

```
(function () {
  const privateMessage = "This variable can't leak into the global scope";
  console.log(privateMessage);
})();
```

> **➼ Exercise**
> Write a short IIFE that logs a message immediately when the script loads, and note in a comment why ES6 modules have largely replaced this pattern in modern code.

JSON JavaScript Object Notation

A lightweight, text-based data format based on JavaScript's own object/array syntax, used constantly for sending and receiving data over the network (as seen with fetch in Chapter 11) and for storing structured data as plain text.

Values / Details

Concept / Value	What it does
JSON.stringify(value)	Converts a JavaScript object/array into a JSON-formatted string.
JSON.parse(string)	Converts a JSON-formatted string back into a real JavaScript object/array.

Example

```
const client = {name: "Selam Traders", plan: "Pro”};
const jsonString = JSON.stringify(client);
console.log(jsonString); // '{"name":"Selam Traders","plan":"Pro"}'

const parsedBack = JSON.parse(jsonString);
console.log(parsedBack.name); // "Selam Traders"
```

> **➼ Exercise**
> Convert one of your client objects into a JSON string using JSON.stringify(), log it, then parse it back into a real object with JSON.parse() and confirm you can still access its properties normally.

localStorage

Let’s a webpage save small pieces of text data directly in the visitor's browser, persisting even after the page is closed and reopened commonly combined with JSON.stringify/parse to save whole objects, like a saved draft of a contact form.

Values / Details

Concept / Value	What it does
localStorage.setItem(key, value)	Saves a string value under a given key (non-string values must be converted with JSON.stringify first).
localStorage.getItem(key)	Retrieves the saved string, or null if nothing was saved under that key.
localStorage.removeItem(key)	Deletes one saved item.

Example

```
const draft = {name: "Selam", message: "Interested in a website”};
localStorage.setItem("quoteDraft", JSON.stringify(draft));

const saved = localStorage.getItem("quoteDraft");
if (saved) {
  const draftData = JSON.parse(saved);
  console.log ("Restored draft:", draftData.name);
}
```

> **➖ Exercise**
> Save a small draft object from your contact form into localStorage using JSON.stringify, then reload the page (or open DevTools' Application tab to inspect it directly) and retrieve it back with JSON.parse to confirm it persisted.

Debouncing (a practical performance pattern)

Some events (like input while typing, or scroll) can fire dozens of times per second. Debouncing delays running your handler until the visitor has PAUSED for a moment, preventing wasted work commonly used for a live search box that shouldn't fetch results on every single keystroke.

Example

```
function debounce(fn, delayMs) {
  let timeoutId;
  return function (...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn(...args), delayMs);
  };
}

const handleSearch = debounce((value) => {
```

```
  console.log ("Searching for:", value);
}, 400);

document.querySelector("#search").addEventListener("input", (e) => {
  handleSearch(e.target.value);
});
```

> **➖ Exercise**
> Add the debounce helper function to your script, wrap a console.log 'searching...' call in it with a 400ms delay, and attach it to a text input's input event confirm by typing quickly that it only logs once you pause, not on every keystroke.

Code quality best practices

A few habits that consistently separate maintainable real-world JavaScript from fragile scripts.

Values / Details

Concept / Value	What it does
Prefer const, then let, never var	Reduces accidental reassignment bugs and scoping confusion (Chapters 2 & 5).
Always use === and !==	Avoids the type-coercion surprises of == and != (Chapter 3).
Give functions and variables clear, descriptive names	calculateDiscountedPrice() beats calc() code is read far more often than it's written.
Keep functions small and focused	A function that does one clear thing is easier to test, reuse, and debug than one that does five things.
Handle errors, don't ignore them	Every fetch and JSON.parse should assume it might fail (Chapters 11 & 13).
Comment the 'why', not the 'what'	Code already shows what it does; comments are most valuable explaining WHY a non-obvious decision was made.

Example

```
// Bad:
function calc(x, y) { return x - x * y; }

// Good:
/**
 * We round to 2 decimals because ETB pricing is always shown
 * to the cent on invoices, per Zolax Tech's finance team.
 */
function calculateDiscountedPrice(price, discountPercent) {
```

```
  const discounted = price - price * (discountPercent / 100);
  return Number(discounted.toFixed(2));
}
```

✏ Exercise

Find one function you wrote in an earlier chapter with a short, unclear name or no comments, and refactor it: give it a clear descriptive name, add a short comment explaining any non-obvious business logic, and confirm it still works exactly as before after the rename.

Chapter Summary Project

🏁 Chapter Project

Perform a full code-quality pass on every JavaScript file in your zolaxtech.com.et project: replace any remaining var with let/const, replace any == with ===, rename at least three unclearly-named functions or variables, add try/catch around any remaining unprotected fetch/JSON.parse calls, add a debounce wrapper to your live search or input validation if you built one, and save at least one piece of form-draft data to localStorage with JSON.stringify/parse, confirming it survives a page reload.

Appendix A: Useful VS Code & DevTools Shortcuts for JavaScript

Shortcut	What it does
Ctrl + S	Save the current file (triggers Live Server auto-refresh).
Ctrl + /	Comment or uncomment the selected line(s) with //.
Alt + Shift + F	Auto-format/indent the whole script.
Ctrl + Space	Trigger autocomplete/IntelliSense suggestions for variables and methods.
Ctrl + D	Select the next occurrence of the current word (handy for renaming a repeated variable).
F12 (on a function name)	Jump straight to that function's definition.
Ctrl + `	Open/close the integrated terminal (for running node script.js).
Right-click page > Inspect	Opens DevTools; use the Console tab to run JS live and the Sources tab to set breakpoints.
Ctrl + Shift + J (in Chrome)	Jumps straight to the DevTools Console from the browser.

Appendix B: Quick Reference Common Patterns Cheat Sheet

Goal	JavaScript
Select an element	const el = document.querySelector('.card');
Select all matches	const els = document.querySelectorAll('.card');
Listen for a click	el.addEventListener('click', (e) => { ... });
Toggle a CSS class	el.classList.toggle('active');
Loop over an array	array.forEach((item) => { ... });
Transform an array	const result = array.map((item) => item * 2);
Filter an array	const result = array.filter((item) => item > 10);
Sum an array	const total = array.reduce((sum, n) => sum + n, 0);
Fetch JSON data	const data = await (await fetch(url)).json();
Guard against a missing value	const email = user?.contact?.email ?? 'N/A';

Appendix C: Final Capstone Project

Using everything from this guide, bring the full multi-page zolaxtech.com.et website (from the companion HTML5 and CSS guides) to life with JavaScript, in Visual Studio Code with Live Server, delivering:

1. A working mobile menu toggle using classList, and a sticky/interactive header behavior driven by a scroll or click event.
2. A fully validated quote/contact form: custom validateForm logic, visible on-page error messages, at least one live blur-based field check, and form.reset() on success all wired through a single submit event listener with event.preventDefault().
3. A dynamic services or portfolio section rendered from a JavaScript array of objects using. map()/.forEach() and DOM methods (createElement, classList, textContent) rather than hard-coded HTML.
4. A 'Latest Updates' or similar feature that uses async/await with fetch to load real or sample data, handles errors with try/catch, and renders the results into the page with a loading state.
5. At least one ES6 class (e.g. Client or Service) with a constructor and at least one method, used to model and render real data on the page.
6. Clean, modern syntax throughout: const/let (no var), strict equality (===), template literals, destructuring, and arrow functions used appropriately.
7. Defensive error handling: try/catch around every fetch/JSON.parse call, and optional chaining (?.) anywhere data might be incomplete.
8. All JavaScript split into logical modules (e.g. validation.js, pricing.js, script.js) using import/export, linked via a single <script type="module">.