MICROSOFT DEVELOPMENT

The Microsoft Coding Codex

Visual Studio Community 2026 + Visual Studio Code 1.130

A complete learn-it, build-it and fix-it manual for Windows and Mac

WINDOWS IDE	CROSS-PLATFORM EDITOR	LANGUAGE PATHS
Visual Studio Community 2026	VS Code 1.130	C# • Python • JavaScript • TypeScript • HTML/CSS • SQL

Prepared for Keith • Codex Edition • 25 August 2026

How to use this codex

READ THIS FIRST This is one manual for two different Microsoft products. Visual Studio Community is the full Windows IDE. Visual Studio Code is the lightweight, cross-platform editor used on Mac, Windows and Linux. Commands that belong to one product are always labelled.

Use the book in three ways. Read Parts I and II in order if you are new to programming. Jump to the Visual Studio or VS Code part when you need an interface command. Follow a language pathway when you want to build something real.

Your goal	Start here
I have never coded	Parts I-II, then C# or Python
I use Windows and want the complete Microsoft IDE	Part III: Visual Studio Community
I use a Mac	Part IV: Visual Studio Code, then C#/Python/Web
I want websites	Part VII: HTML, CSS, JavaScript and TypeScript
I want apps and services	Part V: C# and .NET
I want automation and scripts	Part VI: Python
Something is broken	Parts VIII-IX and Appendix I

Every worked example includes the file or project context. Do not paste a fragment into an unrelated file and expect it to run. When a command differs on your machine, search the printed command name in the live Command Palette or menu; installed versions, profiles and extensions are the final authority.

Product boundary and verification date

- Visual Studio Community 2026: Windows-only full IDE. This edition is aligned to Visual Studio 2026 version 18.9.2, released 25 August 2026.
- Visual Studio Code 1.130: cross-platform editor. Version 1.130 was released 22 July 2026.
- Visual Studio for Mac was retired; the Mac pathway in this book uses Visual Studio Code.
- Menus, icons, previews, licensing, AI availability and extensions can change. Verify live behaviour before relying on a printed instruction.

Contents

PART I

Choose the Right Tool

Install deliberately, understand project containers, and protect your work.

CHAPTER 01

Visual Studio, Visual Studio Code and the terminal

WHY THIS MATTERS Choosing the wrong product causes missing-menu problems, incompatible instructions and wasted setup time.

Core understanding

Visual Studio Community is a full integrated development environment. It owns solutions, projects, builds, debugging, tests, profiling, publishing, NuGet and Git through a project-aware Windows interface.

Visual Studio Code is an editor platform. It becomes a development environment through installed runtimes, compilers and extensions. The integrated terminal runs tools already installed on your computer; VS Code itself is not the Python, .NET or Node.js runtime.

A terminal is a text interface to a shell. A shell interprets commands. A runtime executes a program. A compiler transforms source. An editor changes files. Keep those roles separate when diagnosing failures.

Operational notes

- Purple Visual Studio and blue Visual Studio Code are different products.
- A .sln or .slnx solution is not the same thing as a folder or .code-workspace file.
- Saving, committing and pushing are three different actions.

DONE WHEN Correct product identified | Operating system confirmed | Required runtime known

CHAPTER 02

Install Visual Studio Community on Windows

WHY THIS MATTERS Workloads determine which templates, SDKs, compilers and designers exist in the IDE.

Core understanding

The installer is workload-based. Select only the development areas you need, then add individual components when a project requires them. Common starting workloads are .NET desktop development, ASP.NET and web development, Desktop development with C++, Python development, and Node.js development.

Visual Studio Community is free for individual developers, but organisation use is subject to Microsoft licensing limits. Installation does not prove you are licensed for every workplace scenario.

Working method

1. Download Visual Studio Community from Microsoft and launch the Visual Studio Installer.
2. Select .NET desktop development for the C# pathway in this book. Add ASP.NET and web development for the API project.
3. Keep the recommended .NET SDK, Git tooling and Windows SDK components selected.
4. Install, restart if requested, then open Visual Studio and sign in only if you want account-linked features.
5. Use Tools > Get Tools and Features later to modify the installation.

Operational notes

- Missing template usually means missing workload or SDK.
- Do not install every workload merely because it exists; they consume significant disk space.

DONE WHEN Community edition opens | .NET console template visible | Ctrl+Shift+B can build a sample

CHAPTER 03

Install Visual Studio Code on Mac or Windows

WHY THIS MATTERS VS Code needs a language toolchain as well as the editor.

Core understanding

Install VS Code from the official site, then separately install the runtime for the language you will use: .NET SDK for C#, Python for Python, or Node.js for JavaScript/TypeScript tooling.

Extensions add language intelligence, debugging and test integration. Prefer verified publishers and inspect the exact extension identifier before installation.

Working method

6. Install VS Code and open it once so the operating system registers the application.
7. Open Extensions and install C# Dev Kit for C#, the Microsoft Python extension for Python, or the tooling required by your chosen stack.
8. Open a project folder, not only a loose file. Confirm the folder name in Explorer and terminal path.
9. Open the integrated terminal and run a version check such as dotnet --version, python3 --version, node --version and git --version.
10. If macOS asks whether to trust the downloaded application or folder, verify its origin before approving.

Worked example

TERMINAL

```
dotnet --version
python3 --version
node --version
git --version
```

DONE WHEN Editor installed | Runtime installed | Extension installed | Version command succeeds

CHAPTER 04

Files, folders, projects, solutions and workspaces

WHY THIS MATTERS Code fails when its files are outside the expected project structure or the wrong folder is opened.

Core understanding

A source file contains code. A project defines how files become one output. A solution groups related projects. A folder is a filesystem location. A VS Code workspace may contain one or more folders plus workspace settings.

Dependencies belong to a project or package environment, not to a random source file. Configuration files such as .csproj, package.json, pyproject.toml and .vscode/launch.json tell tools how to work.

Worked example

FOLDER TREE

```
MyApp/
 MyApp.sln
 src/
  MyApp/
   MyApp.csproj
   Program.cs
 tests/
  MyApp.Tests/
   MyApp.Tests.csproj
 README.md
 .gitignore
```

Operational notes

- Open the repository root when Git, tests and tasks must see the whole project.
- Do not rename project files in Finder/File Explorer while the IDE is building.

DONE WHEN Root folder known | Project definition present | Source and tests separated

CHAPTER 05

The safe development loop

WHY THIS MATTERS A repeatable loop makes errors smaller and easier to diagnose.

Core understanding

The basic loop is understand, branch, change, format, build, test, inspect, commit. Each step produces evidence. Skipping directly from idea to push removes the checkpoints that protect you.

Change one coherent behaviour at a time. A small diff can be reviewed, tested and reversed. A large mixed diff hides causes and makes merge conflicts harder.

Working method

11. State the behaviour you want and the acceptance check that will prove it.
12. Create or switch to a feature branch.
13. Make the smallest complete code change.
14. Save and format the changed files.
15. Build or run the relevant command and read the first complete failure.
16. Run focused tests, then the wider test suite.
17. Inspect the Git diff for accidental files, secrets and unrelated formatting.
18. Commit with a message describing the change, then push when ready to share.

DONE WHEN Acceptance check passes | Diff reviewed | Tests pass | Commit contains only intended work

CHAPTER 06

Read errors instead of guessing

WHY THIS MATTERS Most debugging time is wasted by reacting to the last visible line rather than the first useful cause.

Core understanding

A compiler error means the source or project cannot be transformed into a valid output. A runtime exception occurs after the program starts. A failing test means observed behaviour differs from an assertion. A warning is not automatically harmless.

Record the exact command, working directory, complete error, first relevant source location and what changed immediately before the failure. Search the exact error only after understanding those facts.

Working method

19. Stop repeating the same command.
20. Read from the first error, not only the final summary.
21. Confirm file, line, project, configuration and working directory.
22. Reduce the problem to the smallest repeatable case.
23. Change one suspected cause and rerun the same check.
24. Revert the experiment if it did not improve the evidence.

Operational notes

- Error List is a summary; Output often holds the full build log.
- Problems is not the terminal; Debug Console is not the normal shell.

DONE WHEN Exact failure captured | Failure reproduced | One-variable test performed

PART II

Programming Foundations

Learn the ideas that transfer between C#, Python, JavaScript and TypeScript.

CHAPTER 07

Source code, syntax and execution

WHY THIS MATTERS Programming languages differ in punctuation, but the underlying reasoning is shared.

Core understanding

Source code is human-readable instruction text. Syntax is the grammar the language accepts. Semantics are what valid code means. A compiler can reject syntax; tests and users expose incorrect meaning.

C# is normally compiled by the .NET toolchain. Python is interpreted through a Python runtime. JavaScript runs in browsers or a runtime such as Node.js. TypeScript is checked and transpiled to JavaScript.

Comments explain why, constraints and non-obvious decisions. They should not repeat an obvious line or preserve dead code that Git already remembers.

Worked example

MULTIPLE LANGUAGES

```
// C#
Console.WriteLine("Hello");

# Python
print("Hello")

// JavaScript
console.log("Hello");
```

DONE WHEN Language identified | Runtime/compiler identified | Entry point identified

CHAPTER 08

Variables, constants and types

WHY THIS MATTERS A variable gives a value a name; a type constrains what the value can represent and which operations are valid.

Core understanding

Use descriptive names that reflect meaning, not storage. priceInPence is safer than p because the unit is visible. Prefer immutable values when reassignment is unnecessary.

Common types include integer, floating-point number, decimal, Boolean, text, date/time and collections. Money should normally use decimal arithmetic rather than binary floating point.

Null or None represents absence, not an empty string, zero or false. Treat optional values explicitly.

Worked example

C#

```
decimal price = 12.50m;
int quantity = 3;
const decimal VatRate = 0.20m;
decimal total = price * quantity;
bool isLargeOrder = total >= 30m;
string message = $"Total: {total:C}";
```

Practice

Create variables for a song title, duration in seconds, whether it is locked, and its credit cost. Print one formatted sentence.

DONE WHEN Names show meaning and units | Correct numeric type chosen | Optional values handled

CHAPTER 09

Operators and expressions

WHY THIS MATTERS Expressions combine values into new values; precedence mistakes can silently change results.

Core understanding

Arithmetic operators calculate; comparison operators produce Boolean values; logical operators combine conditions; assignment changes state. Parentheses make intended grouping explicit.

Integer division can discard a remainder. String concatenation is not numeric addition. Equality and identity are different concepts in some languages.

Worked example

C#

```
int seconds = 380;
int minutes = seconds / 60;
int remainder = seconds % 60;
bool longTrack = seconds > 300 && seconds <= 600;
Console.WriteLine($"{minutes}:{remainder:D2} | long={longTrack}");
```

Operational notes

- Test boundary values: exactly 0, exactly the limit, one below and one above.
- Never compare secret values or floating-point results casually.

DONE WHEN Grouping explicit | Units consistent | Boundary cases tested

CHAPTER 10

Conditions and branching

WHY THIS MATTERS Branches express decisions; overlapping conditions create unreachable or contradictory behaviour.

Core understanding

An if statement evaluates conditions in order. The first matching branch runs. Place narrow special cases before broad cases, or design mutually exclusive ranges.

A switch or match expression is useful when one value determines a finite set of outcomes. A guard clause exits early when required conditions are not met.

Worked example

C#

```
static string Grade(int score)
{
   if (score is < 0 or > 100)
      throw new ArgumentOutOfRangeException(nameof(score));
   if (score >= 85) return "Distinction";
   if (score >= 70) return "Merit";
   if (score >= 50) return "Pass";
   return "Not yet passed";
}
```

DONE WHEN All inputs accounted for | Invalid input rejected | Boundaries tested

CHAPTER 11

Loops and iteration

WHY THIS MATTERS Loops automate repetition but must have a clear collection, condition or termination rule.

Core understanding

Use foreach/for…of when visiting items. Use for when an index is required. Use while when repetition depends on a condition that changes during the loop.

Avoid modifying a collection while iterating unless the language and algorithm explicitly support it. Infinite loops usually mean the termination state never changes.

Worked example

C#

```
var durations = new[] { 333, 270, 284, 287 };
int total = 0;
foreach (int seconds in durations)
{
   total += seconds;
}
Console.WriteLine($"Album time: {total / 60}:{total % 60:D2}");
```

DONE WHEN Loop terminates | Empty collection works | Accumulator initialized correctly

CHAPTER 12

Functions, parameters and return values

WHY THIS MATTERS Functions make behaviour reusable, testable and nameable.

Core understanding

A function should do one coherent job. Parameters are inputs; the return value is output. Avoid hidden dependence on global state when the needed value can be passed explicitly.

Separate calculation from input/output. A pure calculation is easier to test because the same inputs produce the same result without touching files, time, networks or the screen.

Worked example

C#

```
static string FormatDuration(int seconds)
{
    if (seconds < 0)
        throw new ArgumentOutOfRangeException(nameof(seconds));
    return $"{seconds / 60}:{seconds % 60:D2}";
}

Console.WriteLine(FormatDuration(380)); // 6:20
```

DONE WHEN Name states behaviour | Inputs validated | Return value tested | Side effects isolated

CHAPTER 13

Collections and data structures

WHY THIS MATTERS The right data structure makes operations clear and prevents duplicate or invalid state.

Core understanding

Lists preserve order and allow duplicates. Sets model unique membership. Dictionaries/maps associate keys with values. Queues are first-in-first-out; stacks are last-in-first-out.

Choose based on required operations rather than habit. If lookup by title is central, a dictionary may be clearer than repeatedly scanning a list.

Worked example

C#

```
var tracks = new Dictionary<int, string>
{
   [1] = "I Miss Who I Was",
   [2] = "A Photograph Lies",
   [3] = "The Colour of Before"
};

foreach (var pair in tracks.OrderBy(p => p.Key))
   Console.WriteLine($"{pair.Key:00}. {pair.Value}");
```

DONE WHEN Ordering requirement known | Duplicate policy known | Lookup key stable

CHAPTER 14

Strings, dates, units and culture

WHY THIS MATTERS Text and numbers become unreliable when encoding, locale, timezone or units are implicit.

Core understanding

Use interpolation for readable formatting. Treat user-facing text, identifiers and file paths differently. Normalize only when the domain allows it.

Store instants in UTC when systems cross time zones, then convert for display. Use explicit formats when machines exchange dates. Use structured duration types when available.

Text is Unicode. Count the unit your target system defines: code points, UTF-16 code units, bytes and grapheme clusters can differ.

Worked example

C#

```
DateTimeOffset now = DateTimeOffset.UtcNow;
string machine = now.ToString("O");
string display = now.ToLocalTime().ToString("dd MMMM yyyy HH:mm");
Console.WriteLine(machine);
Console.WriteLine(display);
```

DONE WHEN Units named | Timezone explicit | Machine format separated from display

CHAPTER 15

Errors, exceptions and validation

WHY THIS MATTERS Validation prevents bad state; exceptions report failures that normal control flow cannot complete.

Core understanding

Validate at system boundaries: user input, files, network responses, configuration and database data. Do not catch every exception merely to hide it.

Catch only when you can recover, add meaningful context, translate to a domain error or release resources. Preserve the original exception when rethrowing.

Worked example

C#

```
try
{
    string text = File.ReadAllText("settings.json");
    Console.WriteLine(text);
}
catch (FileNotFoundException ex)
{
    Console.Error.WriteLine($"Settings file missing: {ex.FileName}");
}
catch (UnauthorizedAccessException)
{
    Console.Error.WriteLine("The process cannot read that file.");
}
```

DONE WHEN Boundary validated | Specific failure handled | Failure not silently swallowed

CHAPTER 16

Files, JSON and persistence

WHY THIS MATTERS Persistent data survives the process, so format changes and partial writes must be deliberate.

Core understanding

Use platform path functions instead of joining path strings manually. Decide encoding explicitly when interoperability matters. Write to a temporary file and replace the target for important data.

JSON represents objects, arrays, strings, numbers, booleans and null. It does not inherently carry comments, dates, decimal semantics or schema guarantees.

Worked example

C#

```
using System.Text.Json;

var album = new { Title = "Sloane", Tracks = 20, Locked = true };
var options = new JsonSerializerOptions { WriteIndented = true };
string json = JsonSerializer.Serialize(album, options);
File.WriteAllText("album.json", json);
Console.WriteLine(File.ReadAllText("album.json"));
```

DONE WHEN Path is correct | Encoding understood | Schema/version strategy considered | Write failure handled

CHAPTER 17

Objects, classes and interfaces

WHY THIS MATTERS Object-oriented design groups state with valid behaviour; it should clarify the domain rather than create ceremony.

Core understanding

A class defines a reference type; an object is an instance. Encapsulation protects invariants. Composition combines objects. Inheritance should model a genuine substitutable relationship, not code reuse alone.

An interface defines a capability contract. Dependency injection passes dependencies from outside, making implementations replaceable and tests easier.

Worked example

C#

```
public sealed class Track
{
    public string Title { get; }
    public TimeSpan Duration { get; private set; }

    public Track(string title, TimeSpan duration)
    {
        if (string.IsNullOrWhiteSpace(title)) throw new ArgumentException("Title required");
        if (duration <= TimeSpan.Zero) throw new ArgumentOutOfRangeException(nameof(duration));
        Title = title;
        Duration = duration;
    }
}
```

DONE WHEN Invariant enforced | Public surface minimal | Composition preferred where suitable

CHAPTER 18

Asynchronous work and concurrency

WHY THIS MATTERS Slow I/O should not freeze an interface or block a server thread, but concurrency introduces ordering and cancellation problems.

Core understanding

Async/await expresses non-blocking waits for I/O. It does not automatically make CPU-heavy work faster. Await tasks rather than blocking with .Result or .Wait in asynchronous code.

Support cancellation for work users may abandon. Protect shared mutable state. Assume operations can fail after partial progress.

Worked example

C#

```
using var client = new HttpClient();
using var timeout = new CancellationTokenSource(TimeSpan.FromSeconds(10));
try
{
    string text = await client.GetStringAsync("https://example.com", timeout.Token);
    Console.WriteLine(text.Length);
}
catch (OperationCanceledException)
{
    Console.Error.WriteLine("Request cancelled or timed out.");
}
```

DONE WHEN Task awaited | Cancellation considered | Timeout set | Shared state controlled

CHAPTER 19

Algorithms, complexity and clear code

WHY THIS MATTERS Correct code can still become unusable when work grows faster than the data.

Core understanding

Big-O notation describes growth, not exact speed. A single pass is generally O(n); nested full scans are often $O(n^2)$; dictionary lookup is commonly O(1) average case.

Measure before optimizing. First choose the correct algorithm and data structure, then profile real workloads. Clear code with tests is easier to optimize safely.

Worked example

C#

```
// Avoid scanning the whole list for every lookup.
var byTitle = tracks.ToDictionary(t => t.Title, StringComparer.OrdinalIgnoreCase);
if (byTitle.TryGetValue("Stay", out var track))
    Console.WriteLine(track.Duration);
```

DONE WHEN Expected data size known | Dominant operation known | Measured before micro-optimizing

PART III

Visual Studio Community 2026

Master the complete Windows IDE: solutions, builds, debugging, Git, tests and publishing.

CHAPTER 20

Start window and project creation

WHY THIS MATTERS The start action determines whether Visual Studio creates, clones or opens project metadata.

Core understanding

Create a new project opens the installed template catalogue. Clone a repository downloads Git history. Open a project or solution loads project-system metadata. Open a local folder works without a solution. Continue without code opens the IDE alone.

A template is a project starter. A workload installs groups of templates and tools. If a template is absent, inspect the installer rather than inventing project files.

Working method

25. Choose Create a new project.
26. Filter by C#, Windows and Console, then choose Console App.
27. Name the project, choose a deliberate folder, and keep solution/project naming clear.
28. Choose the latest supported .NET target your deployment permits.
29. Build once before editing to prove the generated baseline works.

DONE WHEN Correct template | Correct target framework | Baseline builds

CHAPTER 21

The Visual Studio screen

WHY THIS MATTERS Each tool window answers a different question; opening the wrong one hides the evidence you need.

Core understanding

Solution Explorer describes project structure. Properties changes the selected object's immediate values. Error List aggregates diagnostics. Output shows detailed logs by source. Terminal runs shell commands. Test Explorer manages tests. Git Changes manages the working tree.

Tool windows can dock, float, pin and auto-hide. Window > Reset Window Layout repairs a broken arrangement without deleting project files.

Operational notes

- Solution Explorer: View > Solution Explorer or Ctrl+Alt+L.
- Output: View > Output or Ctrl+Alt+O.
- Error List: View > Error List or Ctrl+\, E.
- Properties: View > Properties Window or F4.
- Test Explorer: Test > Test Explorer.
- Git Changes: Git > Git Changes.

DONE WHEN Can reopen every panel | Can identify active configuration | Can identify startup project

CHAPTER 22

Solution Explorer and references

WHY THIS MATTERS Selecting the wrong node changes the meaning of Add, Build, Properties and startup commands.

Core understanding

The solution is the top-level workspace. Projects produce apps, libraries, services or test assemblies. Dependencies shows framework, project and package references. Files sit inside or are linked into projects according to project type.

Use Add New Item for a generated project item, Add Existing Item for an existing file, Project Reference for another project, and NuGet for a published package.

Working method

30. Select the intended project before adding an item.
31. Right-click the project and choose Add > New Item.
32. Name the class or configuration file clearly.
33. Check its namespace and build action if relevant.
34. Build to confirm the project includes it correctly.

DONE WHEN Correct project selected | Reference type correct | Startup project correct

CHAPTER 23

Editor, IntelliSense and refactoring

WHY THIS MATTERS Language-aware editing accelerates safe change only when you review the scope and preview.

Core understanding

IntelliSense provides completion, member lists and parameter information. Quick Actions offers fixes and refactorings. Rename updates resolved references. Go to Definition and Find All References reveal dependency direction.

A coloured squiggle may come from live analysis before a build. Confirm actual compiler failures in Error List and Output.

Operational notes

- Quick Actions: Ctrl+.
- Rename: Ctrl+R, Ctrl+R.
- Format document: Ctrl+K, Ctrl+D.
- Format selection: Ctrl+K, Ctrl+F.
- Go to Definition: F12.
- Peek Definition: Alt+F12.
- Find All References: Shift+F12.
- Go to All: Ctrl+T or Ctrl+,.

DONE WHEN Preview reviewed | References checked | Build and tests rerun

CHAPTER 24

Build configurations and startup

WHY THIS MATTERS Build, rebuild, clean, run and debug are different operations.

Core understanding

Build compiles required or out-of-date targets. Rebuild cleans then builds. Clean removes generated outputs for the selected configuration. F5 starts with debugger attachment; Ctrl+F5 starts without it.

Debug and Release are named property sets, not universal guarantees. Confirm optimization, symbols, environment and target through project settings.

Working method

35. Select Debug and the intended platform/target.
36. Confirm the intended startup project or launch profile.
37. Build Solution with Ctrl+Shift+B.
38. Read the first build error in Output > Build.
39. Run without debugging with Ctrl+F5 for a clean behavioural check, then use F5 when inspection is needed.

DONE WHEN Build succeeds | Correct target launches | Output source selected

CHAPTER 25

Visual Studio debugger

WHY THIS MATTERS A debugger turns a running program into observable state; stepping without a question produces noise.

Core understanding

A breakpoint pauses before a bound statement executes. The yellow arrow marks the next statement. Step Over executes a call without entering it; Step Into enters; Step Out returns to the caller.

Locals shows current-scope values. Watch tracks selected expressions. Call Stack shows how execution arrived. Exception Settings controls when exception types break. Modules shows loaded binaries and symbol status.

Working method

40. State the value or branch you expect.
41. Place one breakpoint with F9 immediately before the decision.
42. Start with F5 and confirm the breakpoint binds.
43. Inspect Locals and the selected stack frame.
44. Use F10/F11 deliberately until actual state diverges from expected state.
45. Fix the cause, restart with Ctrl+Shift+F5, and repeat the same path.

Operational notes

- Stop debugging: Shift+F5.
- Run to Cursor: Ctrl+F10.
- Call Stack: Ctrl+Alt+C.
- Breakpoints: Ctrl+Alt+B.

DONE WHEN Question stated | Breakpoint bound | First divergence found | Regression test added

CHAPTER 26

Git inside Visual Studio

WHY THIS MATTERS The Git UI is a view over repository state; it does not replace understanding stage, commit, fetch, pull and push.

Core understanding

Git Changes shows modified files and diffs. Staging selects content for the next snapshot. Commit records locally. Fetch downloads remote references. Pull integrates. Push publishes local commits.

Visual Studio 2026 adds richer multi-file diff, submodule and worktree experiences in current releases. These features do not change Git's underlying model.

Working method

46. Create a named branch from the current correct base.
47. Edit and save, then inspect each diff.
48. Stage only coherent changes and write an imperative commit message.
49. Fetch before integrating remote work.
50. Resolve conflicts in the merge editor, build and test, then commit the resolution.
51. Push only after the local history and working tree are understood.

DONE WHEN Correct branch | Diff reviewed | No secret staged | Tests pass before push

CHAPTER 27

NuGet and dependency management

WHY THIS MATTERS Every dependency adds code, version constraints, licences and supply-chain risk.

Core understanding

NuGet package references belong in project metadata. Restore retrieves packages; build consumes them. A packages.lock.json file can make resolution more repeatable when configured.

Choose actively maintained packages, review owners and licence, avoid unnecessary dependencies, and update in controlled branches with tests.

Worked example

TERMINAL

```
dotnet add package Microsoft.Data.Sqlite
dotnet list package
dotnet list package --outdated
dotnet restore
```

Operational notes

- Package Manager UI and dotnet CLI modify the same project model.
- Do not paste a package name from an untrusted instruction without verifying it.

DONE WHEN Package identity verified | Version recorded | Licence reviewed | Restore and tests pass

CHAPTER 28

Testing, coverage and profiling in Visual Studio

WHY THIS MATTERS Tests prove specified behaviour; profiling measures where time and resources actually go.

Core understanding

Test Explorer discovers tests through installed adapters. Unit tests isolate logic; integration tests exercise boundaries; end-to-end tests validate a user path. Coverage shows executed lines, not correctness.

Use the Performance Profiler for CPU, memory and other supported tools. Measure a representative Release build where optimization matters.

Working method

52. Open Test Explorer and run the smallest relevant tests.
53. Debug one failing test when state inspection is needed.
54. Run the full suite before merge.
55. Define a performance scenario and baseline measurement.
56. Profile, change one bottleneck, then repeat the same measurement.

DONE WHEN Tests deterministic | Failure message useful | Profile based on representative workload

CHAPTER 29

Settings, extensions, AI and publishing

WHY THIS MATTERS IDE convenience must remain subordinate to review, security and repeatability.

Core understanding

Tools > Options configures editor, keyboard, Git, build and other IDE behaviour. Extensions run code in your development environment. AI features can propose edits and commands but cannot verify product intent for you.

Publishing packages an application for a target. Configuration, secrets, database migrations, certificates and hosting are separate deployment concerns.

Working method

57. Review an extension's publisher, identifier, permissions and maintenance before installation.
58. For AI changes, state the scope, inspect every diff and terminal command, run tests and commit only intended files.
59. Create a publish profile for the actual target, using Release configuration.
60. Keep secrets outside source control and provide production configuration through the deployment environment.
61. Smoke-test the published output on a clean target where possible.

DONE WHEN Extension trusted | AI diff reviewed | Secrets externalized | Published artifact tested

PART IV

Visual Studio Code 1.130

Turn the cross-platform editor into a deliberate development environment.

CHAPTER 30

The VS Code workbench

WHY THIS MATTERS VS Code is organized around editors, views, panels, the status bar and command discovery.

Core understanding

The Activity Bar switches Explorer, Search, Source Control, Run and Debug, Extensions and contributed views. The Primary Side Bar hosts the active view. Editor groups hold files. The Panel hosts Terminal, Problems, Output and Debug Console.

The Command Palette is the universal action search. Quick Open searches files. The status bar displays language mode, branch, diagnostics and environment context.

Operational notes

- Command Palette: Shift+Command+P / Ctrl+Shift+P.
- Quick Open: Command+P / Ctrl+P.
- Toggle Side Bar: Command+B / Ctrl+B.
- Toggle Panel: Command+J / Ctrl+J.

DONE WHEN Can reopen each view | Correct language mode visible | Correct folder and branch visible

CHAPTER 31

Folders, workspaces and settings scope

WHY THIS MATTERS VS Code features resolve from the open workspace; a loose file lacks project context.

Core understanding

Open Folder makes one directory the root for Explorer, Search, Git discovery and default terminal location. A multi-root workspace holds several roots in a .code-workspace file.

User settings apply broadly. Workspace settings apply to the project and override user settings. Folder settings can apply within multi-root workspaces. Version .vscode settings only when the team should share them.

Worked example

.VSCODE/SETTINGS.JSON

```
{
 "editor.formatOnSave": true,
 "files.trimTrailingWhitespace": true,
 "editor.rulers": [100]
}
```

DONE WHEN Repository root open | Settings scope intentional | Team settings safe to share

CHAPTER 32

Extensions and Workspace Trust

WHY THIS MATTERS A workspace and its extensions can run tasks, scripts, debuggers and code with your permissions.

Core understanding

Workspace Trust lets you inspect unfamiliar folders in Restricted Mode. Trust only after reviewing the source and configuration. Extensions can contribute commands, language servers, debug adapters and tasks.

Install the minimum tooling required. Disable a suspected extension for the workspace before uninstalling globally when diagnosing conflicts.

Working method

62. Open an unfamiliar repository in Restricted Mode.
63. Inspect README, package scripts, .vscode files and dependency manifests.
64. Verify extension publisher and exact extension ID.
65. Trust only when you intend to run project-controlled code.
66. If behaviour changes, inspect the extension's Output channel and disable it for the workspace.

DONE WHEN Folder source known | Extension identity verified | Trust decision deliberate

CHAPTER 33

Terminal, Problems, Output and Debug Console

WHY THIS MATTERS These four surfaces show different evidence and accept different kinds of input.

Core understanding

Terminal runs a shell. Problems aggregates diagnostics. Output contains selected logs. Debug Console evaluates expressions in an active debug context and displays debugger output.

The terminal normally starts in the workspace root, but every tab has its own shell and current directory. A command copied for PowerShell may not work in zsh or bash.

Worked example

TERMINAL

```
pwd              # macOS/Linux current directory
Get-Location     # PowerShell current directory
ls               # macOS/Linux list
Get-ChildItem    # PowerShell list
git status       # repository state
```

Operational notes

- Toggle terminal: Control+` on Mac, Ctrl+` on Windows.
- Read multiline commands before pressing Return.

DONE WHEN Shell identified | Current directory confirmed | Correct output surface selected

CHAPTER 34

Editing, navigation and refactoring in VS Code

WHY THIS MATTERS VS Code commands depend on the active language service and workspace context.

Core understanding

IntelliSense, definitions, references, rename, quick fixes and formatting come from built-in or extension language services. A syntax highlighter alone does not provide semantic refactoring.

Use Search for textual evidence and Find References for semantic evidence. Preview workspace-wide replacements before applying.

Operational notes

- Find: Command+F / Ctrl+F.
- Workspace Search: Shift+Command+F / Ctrl+Shift+F.
- Quick Fix: Command+. / Ctrl+..
- Rename: F2.
- Definition: F12.
- References: Shift+F12.

DONE WHEN Correct language service active | Refactor preview reviewed | Format and tests rerun

CHAPTER 35

Run and debug configurations

WHY THIS MATTERS A repeatable debug configuration records program, arguments, environment and working directory.

Core understanding

VS Code includes JavaScript/TypeScript debugging; other languages normally need an extension. Simple programs may run from the active file. Complex projects use .vscode/launch.json.

Breakpoints, variables, watch, call stack and stepping follow the same debugging model as Visual Studio. An unbound gray breakpoint means the debug adapter did not resolve it to executable code.

Worked example

.VSCODE/LAUNCH.JSON

```
{
 "version": "0.2.0",
 "configurations": [
  {
   "name": "Debug Python CLI",
   "type": "debugpy",
   "request": "launch",
   "program": "${workspaceFolder}/app.py",
   "console": "integratedTerminal"
  }
 ]
}
```

DONE WHEN Debugger extension installed | Program path correct | Working directory correct | Breakpoint bound

CHAPTER 36

Tasks and repeatable commands

WHY THIS MATTERS Tasks turn build, lint, test and packaging commands into named workspace operations.

Core understanding

A task invokes an external command. It does not replace the tool. Tasks can be grouped as build/test, depend on other tasks and use problem matchers to populate Problems.

Keep shared tasks platform-neutral where possible. If commands differ by shell or operating system, document or separate them explicitly.

Worked example

.VSCODE/TASKS.JSON

```
{
 "version": "2.0.0",
 "tasks": [
  {
   "label": "test",
   "type": "shell",
   "command": "dotnet test",
   "group": "test",
   "problemMatcher": "$msCompile"
  }
 ]
}
```

DONE WHEN Command works manually | Task label clear | Problem matcher appropriate

CHAPTER 37

Git, testing and remote development in VS Code

WHY THIS MATTERS The interface integrates external Git, testing adapters and remote environments without changing their underlying rules.

Core understanding

Source Control and terminal Git operate on the same repository. The Testing view is populated by language/framework extensions. Remote-SSH, Dev Containers, WSL and Tunnels move execution context away from the local machine.

When remote, extensions may install locally, remotely or both. Confirm where the terminal and debugger are running before diagnosing missing files or runtimes.

Working method

67. Confirm the repository and branch in Source Control.
68. Inspect diff, stage and commit deliberately.
69. Install the test extension for the project's framework and confirm discovery from the repository root.
70. For remote work, read the remote indicator and open a terminal to verify hostname, path and tool versions.
71. Run tests in the same environment that runs the application.

DONE WHEN Git executable available | Tests discovered | Execution environment confirmed

CHAPTER 38

AI and agents in VS Code

WHY THIS MATTERS An agent can accelerate changes, but authority remains with the reviewed diff, tests and repository policy.

Core understanding

Availability depends on installed features, sign-in, plan and organisation policy. An agent may edit multiple files, run commands or use tools subject to permissions. This ChatGPT conversation is not automatically the agent inside VS Code.

Prompt with the desired behaviour, boundaries, files allowed to change and verification command. Never use custom instructions as a security control.

Working method

72. Create a clean branch and record the baseline test result.
73. Give the agent one bounded outcome and explicit exclusions.
74. Inspect proposed commands before approval.
75. Review every changed file and reject unrelated edits.
76. Run tests and security checks independently.
77. Commit only the verified result.

DONE WHEN Scope bounded | Commands reviewed | Diff reviewed | Tests independently run

PART V

C# and .NET

Build strongly typed applications in either Visual Studio Community or VS Code.

CHAPTER 39

Create and run a .NET console app

WHY THIS MATTERS A console app is the cleanest place to learn C# syntax, project structure, builds and debugging.

Core understanding

In Visual Studio choose Console App. In VS Code use .NET: New Project with C# Dev Kit or the dotnet CLI. Both produce a project that the same .NET SDK can build.

Program.cs may use top-level statements. The .csproj file declares target framework, output type, nullable analysis and other project settings.

Working method

78. Create a folder named DurationTool.
79. Run dotnet new console or create the Console App template.
80. Replace Program.cs with the example.
81. Run dotnet run or Ctrl+F5.
82. Set a breakpoint on the calculation and inspect the parsed input.

Worked example

PROGRAM.CS

```
Console.Write("Seconds: ");
string? input = Console.ReadLine();
if (!int.TryParse(input, out int seconds) || seconds < 0)
{
    Console.Error.WriteLine("Enter a non-negative whole number.");
    return;
}
Console.WriteLine($"{seconds / 60}:{seconds % 60:D2}");
```

DONE WHEN dotnet build succeeds | Valid and invalid input tested | Breakpoint inspected

CHAPTER 40

C# types, nullability and records

WHY THIS MATTERS C#'s type system catches invalid combinations before runtime when the model is precise.

Core understanding

Value types include numeric types, bool, char, structs and enums. Classes, arrays, delegates and strings are reference types. Nullable reference analysis warns when code may dereference null.

Records are concise data-centric types with value-based equality. Use classes for identity and mutable lifecycle; use records for immutable values and messages where appropriate.

Worked example

C#

```
public enum TrackStatus { Draft, Locked, Released }

public sealed record Track(
   int Number,
   string Title,
   TimeSpan Duration,
   TrackStatus Status);

Track track = new(1, "I Miss Who I Was", TimeSpan.FromSeconds(333), TrackStatus.Locked);
```

DONE WHEN Nullability enabled | Enum models finite state | Record/class choice intentional

CHAPTER 41

LINQ and collection transformations

WHY THIS MATTERS LINQ expresses filtering, mapping, grouping and aggregation while preserving a readable data pipeline.

Core understanding

Where filters; Select transforms; OrderBy sorts; GroupBy groups; Any and All answer predicates; Sum and Aggregate combine values. Many queries are deferred until enumerated.

Do not hide expensive repeated enumeration. Materialize with ToList when you need a stable snapshot, then name the reason.

Worked example

C#

```
var longTracks = tracks
    .Where(t => t.Duration > TimeSpan.FromMinutes(5))
    .OrderByDescending(t => t.Duration)
    .Select(t => new { t.Title, Seconds = (int)t.Duration.TotalSeconds })
    .ToList();

foreach (var item in longTracks)
    Console.WriteLine($"{item.Title}: {item.Seconds}s");
```

DONE WHEN Query readable | Enumeration count understood | Empty result handled

CHAPTER 42

C# classes, interfaces and dependency injection

WHY THIS MATTERS Separating domain logic from storage and external services keeps programs testable.

Core understanding

Define interfaces at boundaries that genuinely need substitution. Pass collaborators through constructors. Keep domain objects free from UI and database details where practical.

Avoid service locator and global mutable singletons. A dependency injection container assembles objects; it does not remove the need for clear ownership and lifetimes.

Worked example

C#

```
public interface ITrackStore
{
    Task<IReadOnlyList<Track>> LoadAsync(CancellationToken ct);
}

public sealed class AlbumService(ITrackStore store)
{
    public async Task<TimeSpan> TotalDurationAsync(CancellationToken ct)
    {
        var tracks = await store.LoadAsync(ct);
        return TimeSpan.FromSeconds(tracks.Sum(t => t.Duration.TotalSeconds));
    }
}
```

DONE WHEN Boundary explicit | Dependency passed in | Lifetime and cancellation considered

CHAPTER 43

C# files, JSON and SQLite

WHY THIS MATTERS Local applications often progress from a JSON file to a database; each has different consistency guarantees.

Core understanding

System.Text.Json handles common JSON serialization. Microsoft.Data.Sqlite provides SQLite access through NuGet. Use parameters, transactions and migrations/schema versioning rather than string-built SQL.

Dispose connections and commands. Do not store passwords or tokens in source-controlled JSON.

Worked example

C#

```
using Microsoft.Data.Sqlite;

await using var connection = new SqliteConnection("Data Source=tracks.db");
await connection.OpenAsync();
var command = connection.CreateCommand();
command.CommandText = "INSERT INTO Track (Title, Seconds) VALUES ($title, $seconds)";
command.Parameters.AddWithValue("$title", "Bringing Forever Home");
command.Parameters.AddWithValue("$seconds", 380);
await command.ExecuteNonQueryAsync();
```

DONE WHEN Parameterized SQL | Connection disposed | Schema versioned | Secrets excluded

CHAPTER 44

C# unit tests

WHY THIS MATTERS A unit test names one behaviour, supplies controlled inputs and makes a precise assertion.

Core understanding

Arrange creates state, Act performs behaviour, Assert checks the outcome. Test public behaviour, boundary cases and failures. Avoid tests that merely duplicate the implementation.

A failing test should explain expected and actual values. Deterministic tests do not depend on local time, network, random order or shared files without control.

Worked example

XUNIT C#

```
public class DurationFormatterTests
{
   [Theory]
   [InlineData(0, "0:00")]
   [InlineData(380, "6:20")]
   public void Formats_seconds(int seconds, string expected)
   {
      Assert.Equal(expected, DurationFormatter.Format(seconds));
   }

   [Fact]
   public void Rejects_negative_values()
   {
      Assert.Throws<ArgumentOutOfRangeException>(() => DurationFormatter.Format(-1));
   }
}
```

DONE WHEN Happy path | Boundary | Invalid input | Failure message useful

CHAPTER 45

ASP.NET Core minimal API

WHY THIS MATTERS A web API exposes behaviour over HTTP and therefore needs validation, status codes, cancellation and security boundaries.

Core understanding

Routes map HTTP methods and paths to handlers. Request models are untrusted input. Return appropriate status codes and avoid exposing internal exception detail.

Configuration and secrets come from providers such as environment variables and secret stores. Authentication and authorization are distinct from input validation.

Worked example

ASP.NET CORE C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/tracks/{seconds:int}", (int seconds) =>
{
    if (seconds < 0) return Results.BadRequest(new { error = "seconds must be non-negative" });
    return Results.Ok(new { seconds, formatted = $"{seconds / 60}:{seconds % 60:D2}" });
});

app.Run();
```

DONE WHEN Input validated | Status code appropriate | Secrets external | Endpoint tested

CHAPTER 46

Windows desktop applications

WHY THIS MATTERS Visual Studio Community supports Windows UI frameworks; the UI thread and event model introduce new constraints.

Core understanding

Windows Forms is approachable and designer-driven. WPF uses XAML, data binding and a richer presentation model. WinUI is another modern Windows application path. Choose based on deployment target, team knowledge and framework lifecycle.

Keep business logic outside button-click handlers. Long work should be asynchronous and cancellation-aware so the interface remains responsive.

Worked example

WINDOWS FORMS C#

```
private async void LoadButton_Click(object sender, EventArgs e)
{
    LoadButton.Enabled = false;
    try
    {
        TracksList.DataSource = await _trackService.LoadAsync(CancellationToken.None);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Load failed");
    }
    finally
    {
        LoadButton.Enabled = true;
    }
}
```

DONE WHEN UI stays responsive | Logic separated | Errors shown safely | Controls restored

PART VI

Python

Use VS Code for portable scripts, automation, data and services.

CHAPTER 47

Python setup and virtual environments

WHY THIS MATTERS A virtual environment isolates one project's packages from global Python and other projects.

Core understanding

Install a supported Python runtime, then select the interpreter in VS Code. python may be python3 on macOS. The interpreter shown in the status bar must match the environment used by the terminal and debugger.

Create .venv inside the project, activate it for terminal work and exclude it from Git. Record dependencies in pyproject.toml or a requirements file according to the project's tooling.

Worked example

TERMINAL

```
python3 -m venv .venv
source .venv/bin/activate      # macOS/Linux
.venv\Scripts\Activate.ps1    # Windows PowerShell
python -m pip install --upgrade pip
python -m pip list
```

DONE WHEN Interpreter selected | Virtual environment active | .venv ignored by Git

CHAPTER 48

Python syntax, values and control flow

WHY THIS MATTERS Python uses indentation as syntax, dynamic types at runtime and a compact expression model.

Core understanding

Names refer to objects. Type hints document and enable static analysis but do not enforce values at runtime by themselves. Indentation must be consistent.

Use enumerate for index plus item, zip for parallel iteration and comprehensions for simple transformations. Prefer an ordinary loop when logic becomes difficult to read.

Worked example

PYTHON

```
def format_duration(seconds: int) -> str:
    if seconds < 0:
        raise ValueError("seconds must be non-negative")
    minutes, remainder = divmod(seconds, 60)
    return f"{minutes}:{remainder:02d}"

for seconds in [333, 270, 380]:
    print(format_duration(seconds))
```

DONE WHEN Indentation consistent | Type hints meaningful | Invalid input tested

CHAPTER 49

Python collections, functions and modules

WHY THIS MATTERS Modules organize code; collection choices express ordering, uniqueness and lookup.

Core understanding

Lists are ordered and mutable; tuples are ordered and immutable; sets represent uniqueness; dictionaries map keys to values. Functions are first-class objects and can accept keyword arguments.

A module is a .py file. A package is an importable directory structure. Use if __name__ == '__main__' to separate reusable definitions from script execution.

Worked example

PYTHON

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Track:
    title: str
    seconds: int

def total_seconds(tracks: list[Track]) -> int:
    return sum(track.seconds for track in tracks)

if __name__ == "__main__":
    tracks = [Track("Stay", 302), Track("Crown", 282)]
    print(total_seconds(tracks))
```

DONE WHEN Import boundary clear | Execution guard present | Collection matches semantics

CHAPTER 50

Python exceptions, files and JSON

WHY THIS MATTERS Context managers guarantee resource cleanup even when an exception occurs.

Core understanding

Use with for files and other context-managed resources. Catch specific exceptions. pathlib provides cross-platform path operations.

json.load parses a stream; json.loads parses text. Validate required keys and types after parsing because syntactically valid JSON can still violate the application's schema.

Worked example

PYTHON

```
from pathlib import Path
import json

path = Path("album.json")
try:
    with path.open("r", encoding="utf-8") as handle:
        album = json.load(handle)
    title = str(album["title"])
except FileNotFoundError:
    raise SystemExit(f"Missing file: {path.resolve()}")
except (json.JSONDecodeError, KeyError) as exc:
    raise SystemExit(f"Invalid album data: {exc}")

print(title)
```

DONE WHEN Path resolved | Encoding explicit | Specific exceptions | Schema validated

CHAPTER 51

Python classes, dataclasses and protocols

WHY THIS MATTERS Python supports object-oriented design without requiring every piece of state to become a class.

Core understanding

Use dataclasses for data-rich objects. Use properties or validation functions to protect invariants. Protocols describe structural capabilities for type checkers.

Prefer composition. Dunder methods such as __str__ and __eq__ should follow Python conventions and remain unsurprising.

Worked example

PYTHON

```
from dataclasses import dataclass
from typing import Protocol

class TrackStore(Protocol):
    def load(self) -> list["Track"]: ...

@dataclass(frozen=True)
class Track:
    title: str
    seconds: int

    def __post_init__(self) -> None:
        if not self.title.strip() or self.seconds <= 0:
            raise ValueError("valid title and positive duration required")
```

DONE WHEN Class earns its existence | Invariant checked | Protocol describes real boundary

CHAPTER 52

Python testing and debugging

WHY THIS MATTERS Tests and the debugger should reproduce the same environment and interpreter as normal execution.

Core understanding

The standard library unittest framework needs no third-party dependency; pytest is a popular optional alternative. Keep fixtures small and isolate filesystem tests in temporary directories.

In VS Code, select the interpreter, configure tests, set a breakpoint and start the Python debugger. If imports differ, compare working directory and sys.path.

Worked example

PYTHON UNITTEST

```
import unittest
from app import format_duration

class FormatDurationTests(unittest.TestCase):
    def test_formats_380_seconds(self):
        self.assertEqual("6:20", format_duration(380))

    def test_rejects_negative(self):
        with self.assertRaises(ValueError):
            format_duration(-1)

if __name__ == "__main__":
    unittest.main()
```

DONE WHEN Correct interpreter | Tests discovered | Temporary resources isolated

CHAPTER 53

Python command-line applications and SQLite

WHY THIS MATTERS A useful CLI separates argument parsing, domain logic and persistence.

Core understanding

argparse provides standard help and validation. sqlite3 is included in Python's standard library. Parameterize SQL and commit a transaction only after all related statements succeed.

Return useful exit codes: zero for success and non-zero for failure. Print user output to stdout and diagnostics to stderr.

Worked example

PYTHON

```
import argparse
import sqlite3

parser = argparse.ArgumentParser()
parser.add_argument("title")
parser.add_argument("seconds", type=int)
args = parser.parse_args()

with sqlite3.connect("tracks.db") as db:
    db.execute("CREATE TABLE IF NOT EXISTS track (title TEXT NOT NULL, seconds INTEGER NOT NULL)")
    db.execute("INSERT INTO track VALUES (?, ?)", (args.title, args.seconds))

print("Track saved")
```

DONE WHEN --help works | Invalid input rejected | SQL parameterized | Exit status meaningful

PART VII

Web Development

Build browser interfaces with HTML, CSS, JavaScript and TypeScript.

CHAPTER 54

How the web fits together

WHY THIS MATTERS A browser application combines document structure, presentation, behaviour and network requests.

Core understanding

HTML describes meaning and structure. CSS controls presentation. JavaScript changes behaviour and state. HTTP carries requests and responses. A server may generate HTML, expose an API or serve static files.

The browser enforces security boundaries such as same-origin policy and content security rules. Client code is visible to users; never embed secrets in it.

Worked example

FOLDER TREE

```
project/
  index.html
  styles.css
  app.js
  assets/
  README.md
```

DONE WHEN Structure separated | No client secret | Local server used for module/network features

CHAPTER 55

Semantic HTML and accessible structure

WHY THIS MATTERS Semantic elements give browsers, search engines and assistive technology a reliable document outline.

Core understanding

Use headings in order, landmark elements for regions, labels for form controls and buttons for actions. A div is not a button. Alternative text communicates an image's purpose when the image is meaningful.

Validate markup, test keyboard navigation and preserve visible focus. Accessibility is part of correctness, not decorative polish.

Worked example

HTML

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <title>Track Library</title>
 <link rel="stylesheet" href="styles.css">
</head>
<body>
 <main>
  <h1>Track Library</h1>
  <form id="track-form">
   <label for="title">Title</label>
   <input id="title" name="title" required>
   <button type="submit">Add track</button>
  </form>
  <ul id="tracks" aria-live="polite"></ul>
 </main>
 <script type="module" src="app.js"></script>
</body>
</html>
```

DONE WHEN Language set | Labels connected | Keyboard usable | Heading structure valid

CHAPTER 56

CSS layout and responsive design

WHY THIS MATTERS Modern CSS uses the cascade, box model, flexbox and grid to adapt without hard-coded screen assumptions.

Core understanding

Specificity and source order decide which declaration wins. Use classes and a small design system rather than escalating selectors. Flexbox is one-dimensional; Grid is two-dimensional.

Build mobile-first, use relative units, constrain line length and test zoom. Respect reduced-motion preferences.

Worked example

CSS

```
:root {
  --ink: #17212b;
  --accent: #2e74b5;
  --surface: #f4f6f9;
}
* { box-sizing: border-box; }
body { margin: 0; color: var(--ink); font: 1rem/1.5 system-ui, sans-serif; }
main { width: min(70rem, 100% - 2rem); margin-inline: auto; }
.track-grid { display: grid; gap: 1rem; grid-template-columns: repeat(auto-fit, minmax(15rem, 1fr)); }
button:focus-visible { outline: 3px solid var(--accent); outline-offset: 2px; }
@media (prefers-reduced-motion: reduce) { * { scroll-behavior: auto !important; } }
```

DONE WHEN Box sizing set | Responsive without device assumptions | Focus visible | Zoom tested

CHAPTER 57

JavaScript fundamentals and the DOM

WHY THIS MATTERS Browser JavaScript reacts to events and updates a document object model; careless HTML insertion creates security risk.

Core understanding

Use const by default and let for reassignment. Prefer strict equality. Functions are values. Arrays and objects are reference-based. The DOM exposes elements, attributes, events and text.

Use textContent for untrusted text. Do not insert user input through innerHTML. Validate at both client and server boundaries.

Worked example

JAVASCRIPT

```
const form = document.querySelector("#track-form");
const titleInput = document.querySelector("#title");
const list = document.querySelector("#tracks");

form.addEventListener("submit", (event) => {
 event.preventDefault();
 const title = titleInput.value.trim();
 if (!title) return;
 const item = document.createElement("li");
 item.textContent = title;
 list.append(item);
 form.reset();
 titleInput.focus();
});
```

DONE WHEN Selectors checked | Event default handled | Untrusted text not inserted as HTML

CHAPTER 58

Promises, async/await and fetch

WHY THIS MATTERS Network calls are asynchronous and can fail through transport, timeout, HTTP status or invalid data.

Core understanding

fetch resolves for HTTP error statuses; inspect response.ok. Parse the expected content type and handle cancellation. Keep loading, success, empty and error states visible to users.

Use try/catch around awaited work. A finally block restores interface state. Avoid starting duplicate requests accidentally.

Worked example

JAVASCRIPT

```
async function loadTracks(signal) {
  const response = await fetch("/api/tracks", { signal });
  if (!response.ok) {
    throw new Error(`Request failed: ${response.status}`);
  }
  const data = await response.json();
  if (!Array.isArray(data)) throw new Error("Invalid response shape");
  return data;
}

const controller = new AbortController();
loadTracks(controller.signal)
  .then(renderTracks)
  .catch(error => showError(error.message));
```

DONE WHEN HTTP status checked | Data shape checked | Cancellation/error UI considered

CHAPTER 59

TypeScript and tooling

WHY THIS MATTERS TypeScript catches many JavaScript mistakes before execution and documents contracts across a codebase.

Core understanding

TypeScript adds static types then emits JavaScript. Interfaces and type aliases describe shapes. Narrow unknown values before use. Avoid any when the actual shape can be modelled.

tsconfig.json controls checking and output. Use strict mode for new projects. The browser still runs emitted JavaScript, not TypeScript directly.

Worked example

TYPESCRIPT

```
interface Track {
 id: number;
 title: string;
 seconds: number;
}

function formatTrack(track: Track): string {
 const minutes = Math.floor(track.seconds / 60);
 const seconds = String(track.seconds % 60).padStart(2, "0");
 return `${track.title} — ${minutes}:${seconds}`;
}
```

DONE WHEN strict enabled | Unknown input narrowed | Generated output excluded from source edits

CHAPTER 60

Node.js, npm and package scripts

WHY THIS MATTERS Node.js runs JavaScript outside the browser; npm manages packages and reproducible project commands.

Core understanding

package.json records metadata, scripts and dependency ranges. package-lock.json records a resolved dependency graph. npm scripts run project-local tools without assuming global installation.

Review lifecycle scripts and dependency provenance. Do not commit node_modules. Use supported Node versions and controlled updates.

Worked example

PACKAGE.JSON EXAMPLE

```
{
 "name": "track-library",
 "private": true,
 "type": "module",
 "scripts": {
  "dev": "vite",
  "build": "vite build",
  "test": "vitest run"
 },
 "devDependencies": {
  "vite": "^7.0.0",
  "vitest": "^3.0.0"
 }
}
```

Operational notes

- Versions above are examples, not a direction to install without checking current compatibility.
- Run npm audit as evidence, then assess exploitability and update impact rather than treating the count alone as a decision.

DONE WHEN Package identity verified | Lockfile committed | Scripts understood | node_modules ignored

PART VIII

Data, Git and Collaboration

Store information safely and keep a reviewable history.

CHAPTER 61

Relational data and SQL

WHY THIS MATTERS A database protects relationships and enables queries only when schema and transactions are designed deliberately.

Core understanding

Tables contain rows and typed columns. A primary key identifies a row. A foreign key protects a relationship. Constraints encode valid state. Indexes accelerate specific lookups at write and storage cost.

SELECT reads, INSERT creates, UPDATE changes and DELETE removes. Always constrain destructive statements and preview the target rows. Use parameters for values; parameterization does not make arbitrary table or column names safe.

Worked example

SQL

```
CREATE TABLE track (
  id INTEGER PRIMARY KEY,
  album_id INTEGER NOT NULL,
  title TEXT NOT NULL,
  seconds INTEGER NOT NULL CHECK (seconds > 0),
  track_number INTEGER NOT NULL,
  UNIQUE (album_id, track_number),
  FOREIGN KEY (album_id) REFERENCES album(id)
);

SELECT title, seconds
FROM track
WHERE seconds >= 300
ORDER BY seconds DESC;
```

DONE WHEN Primary key | Constraints | Parameterized values | Transaction boundary understood

CHAPTER 62

Git mental model and everyday commands

WHY THIS MATTERS Git records snapshots through a graph of commits; the working tree, index and history are different states.

Core understanding

The working tree is current disk content. The index/staging area is the proposed next snapshot. HEAD is the current checked-out commit. A branch name points into commit history.

Status tells you state. Diff shows changes. Add stages. Commit records. Log shows history. Restore, reset and revert have different destructive implications; prefer a new revert commit for shared history.

Worked example

GIT

```
git status
git switch -c feature/duration-format
git diff
git add src/DurationFormatter.cs tests/DurationFormatterTests.cs
git diff --staged
git commit -m "Add duration formatter"
git fetch --prune
git push -u origin feature/duration-format
```

DONE WHEN Status understood | Staged diff reviewed | Commit coherent | Remote target correct

CHAPTER 63

Branches, merges and conflicts

WHY THIS MATTERS Branches isolate lines of work; conflicts are competing edits requiring a human decision.

Core understanding

Merge combines histories. Rebase replays commits onto a new base and rewrites their identities. Use the team's policy and avoid rebasing commits other people already depend on.

A conflict marker is not a choice to keep both blindly. Understand the intended final behaviour, edit the file, remove markers, build, test, stage and complete the merge or rebase.

Working method

83. Commit or safely stash current work.
84. Fetch remote state.
85. Integrate the correct base according to team policy.
86. Open each conflict and understand both sides plus surrounding code.
87. Produce the intended combined result and remove all markers.
88. Build and test before staging the resolution.
89. Complete the merge/rebase and inspect history.

DONE WHEN No conflict markers | Tests pass | History matches policy

CHAPTER 64

Repository hygiene and collaboration

WHY THIS MATTERS A repository should let another person understand, build, test and change the project without hidden machine state.

Core understanding

README explains purpose and first successful run. .gitignore excludes generated and local-only files. Lockfiles record dependencies. CONTRIBUTING explains workflow. CI repeats checks on a clean machine.

Commit source, configuration examples and migrations. Do not commit credentials, personal data, build outputs, virtual environments or editor caches unless the project explicitly requires them.

Worked example

.GITIGNORE

```
# .gitignore excerpt
**/bin/
**/obj/
.vs/
.vscode/*.local.json
.venv/
__pycache__/
node_modules/
.env
*.user
*.suo
```

DONE WHEN Clean clone can build | README tested | Secrets absent | Generated files ignored

PART IX

Quality, Security and Delivery

Prove behaviour, diagnose failures, protect users and ship repeatably.

CHAPTER 65

Testing strategy

WHY THIS MATTERS Different tests catch different failures; one giant end-to-end suite is slow and fragile, while unit tests alone miss integration.

Core understanding

Unit tests target isolated logic. Integration tests verify boundaries such as databases or HTTP. Contract tests verify interface expectations. End-to-end tests follow important user paths. Smoke tests prove a deployed system starts and performs critical actions.

Test observable behaviour, boundaries and previous defects. A test suite must remain deterministic, isolated enough to run repeatedly and fast enough to use before merge.

Working method

90. Write acceptance examples before implementation.
91. Put calculations and validation under unit tests.
92. Test database, filesystem and HTTP boundaries with controlled fixtures.
93. Add a small number of high-value end-to-end paths.
94. Run focused tests while editing and the full suite before merge.
95. When fixing a bug, add a regression test that failed before the fix.

DONE WHEN Normal path | Boundary | Failure path | Regression | Deterministic run

CHAPTER 66

Systematic debugging

WHY THIS MATTERS Debugging is controlled comparison between expected and observed state.

Core understanding

Start with a reproducible symptom and a minimal failing case. Use logs for event history, a debugger for live state, tests for repeatability and version control for change isolation.

Binary search through commits, inputs or code paths can locate the first bad boundary. A fix without a causal explanation may merely move the symptom.

Working method

96. Write exact reproduction steps and expected/actual result.
97. Confirm environment, version, configuration and input.
98. Reduce the failing surface.
99. Form one falsifiable hypothesis.
100. Add observation at the boundary or use one breakpoint.
101. Run the same reproduction and compare evidence.
102. Fix the earliest causal divergence and add a regression test.
103. Remove temporary diagnostics and verify the full scenario.

DONE WHEN Cause explained | Regression test | Temporary logging removed | Full scenario passes

CHAPTER 67

Secure coding essentials

WHY THIS MATTERS Security failures often begin where untrusted input crosses into commands, queries, files, HTML or authorization decisions.

Core understanding

Validate input by allowed shape and range, encode output for its context, parameterize queries, avoid shell construction, enforce least privilege and keep dependencies current.

Authentication proves identity. Authorization decides permission. Encryption protects data in transit or at rest; it does not make malicious input safe. Logging must avoid secrets and unnecessary personal data.

Operational notes

- Never place API keys, passwords or connection strings in committed source.
- Do not concatenate user input into SQL, shell commands, file paths or HTML.
- Normalize and constrain file paths before access; prevent traversal outside the intended root.
- Use platform cryptography libraries; do not invent encryption.
- Review dependency advisories and update through tested branches.

DONE WHEN Trust boundary listed | Input constrained | Output encoded | Authorization tested | Secrets external

CHAPTER 68

Logging, configuration and observability

WHY THIS MATTERS A production failure cannot be diagnosed if the program emits no structured evidence or if every environment behaves differently.

Core understanding

Logs describe events. Metrics summarize numerical behaviour. Traces connect work across components. Include correlation identifiers, severity and useful context without leaking sensitive data.

Configuration varies by environment; code should not. Validate required configuration at startup and fail clearly. Use environment-specific providers and secret stores.

Worked example

STRUCTURED C# LOGGING

```
logger.LogInformation(
  "Track {TrackId} locked by {ActorId} at {Timestamp}",
  track.Id, actor.Id, DateTimeOffset.UtcNow);
```

DONE WHEN Startup configuration validated | Logs structured | Secrets excluded | Correlation available

CHAPTER 69

Performance and profiling

WHY THIS MATTERS Optimization should follow measurement under representative conditions.

Core understanding

Define the user-visible performance requirement, collect a baseline, profile the bottleneck, change one cause and remeasure. CPU, allocation, disk, network and database wait require different tools.

Caching trades freshness and memory for speed. Concurrency may increase throughput but also contention and failure complexity. A faster wrong answer is not an improvement.

Working method

104. Select one scenario, data size and environment.
105. Warm up if the runtime uses just-in-time compilation.
106. Measure latency, throughput and resource use.
107. Profile to locate dominant work.
108. Optimize the measured cause, preserve correctness tests, and remeasure.
109. Document the before/after evidence and tradeoff.

DONE WHEN Baseline | Profile | One change | Same measurement | Correctness preserved

CHAPTER 70

Build, package and deployment

WHY THIS MATTERS A successful local run is not a deployment; the target needs the correct artifact, runtime, configuration and data changes.

Core understanding

Build produces outputs. Package creates a distributable unit. Publish prepares for a target. Deploy transfers and activates. Release is the user-facing availability decision.

Automate repeatable steps in CI. Keep builds deterministic, sign artifacts where required, scan dependencies, run tests and retain version metadata.

Working method

110. Build a clean Release configuration.
111. Run unit, integration and packaging checks.
112. Create the target-specific publish/package output.
113. Provide configuration and secrets through the environment.
114. Apply database migrations with a rollback/recovery plan.
115. Deploy to a staging environment and run smoke tests.
116. Release gradually where risk warrants it, monitor, then confirm or roll back.

DONE WHEN Versioned artifact | Clean build | Config external | Smoke test | Rollback plan

CHAPTER 71

Documentation and code review

WHY THIS MATTERS Maintainable code communicates purpose, boundaries and verification to the next person—including future you.

Core understanding

A useful README answers what, why, prerequisites, first run, test, build, configuration and troubleshooting. Public APIs need contract documentation. Architecture decisions should record context, decision and consequences.

Review for correctness, tests, security, maintainability, performance and scope. Ask whether the code can be simpler, not whether it matches personal style.

Operational notes

- Describe why a non-obvious constraint exists.
- Keep examples executable and update them with code changes.
- Separate required change from optional suggestion in reviews.

DONE WHEN Fresh-clone instructions work | Public contract documented | Review comments resolved or recorded

PART X

Complete Projects

Apply the whole workflow: plan, create, debug, test, version and deliver.

CHAPTER 72

Project 1 — C# album duration calculator

WHY THIS MATTERS This project combines parsing, records, collections, LINQ, JSON, tests and command-line interaction.

Core understanding

Create a solution with AlbumTool and AlbumTool.Tests projects. Model a Track record, validate duration, load tracks from JSON, calculate totals and print mm:ss or hh:mm:ss.

Acceptance: invalid JSON produces a clear non-zero failure; negative duration is rejected; an empty album reports zero; the selected album total is reproducible by tests.

Working method

117. Create the solution and console/test projects; add a project reference from tests to the app.
118. Write failing tests for FormatDuration and total calculation.
119. Implement Track, Album and pure calculation functions.
120. Add JSON loading with specific error messages.
121. Add command-line path argument and useful --help text.
122. Debug one invalid record, then run the full suite.
123. Initialize Git, commit in coherent stages and publish a Release build.

Worked example

CORE C#

```
public sealed record Track(string Title, int Seconds);

public static TimeSpan TotalDuration(IEnumerable<Track> tracks)
{
    ArgumentNullException.ThrowIfNull(tracks);
    long total = 0;
    foreach (Track track in tracks)
    {
        if (track.Seconds <= 0) throw new InvalidDataException($"Invalid duration: {track.Title}");
        total += track.Seconds;
    }
    return TimeSpan.FromSeconds(total);
}
```

DONE WHEN Tests pass | Invalid input clear | README first run works | Release artifact produced

CHAPTER 73

Project 2 — Python file organiser

WHY THIS MATTERS This project teaches pathlib, CLI design, dry runs, logging, tests and safe filesystem changes.

Core understanding

The program groups files by extension without overwriting. A --dry-run option prints planned moves. Hidden files and folders are skipped by default. Every move is logged.

Acceptance: running twice is safe; filename collisions are handled; the target cannot escape the selected root; tests use a temporary directory.

Working method

124. Create .venv, app.py, tests and README.
125. Write a pure plan_moves function returning source/destination pairs.
126. Test empty folder, mixed extensions, collisions and nested folders.
127. Add argparse options for root and --dry-run.
128. Apply moves only after displaying the plan and validating destinations.
129. Run in dry mode on a copied sample folder before real data.
130. Commit and tag the first working version.

Worked example

PYTHON CORE

```
from pathlib import Path

def plan_moves(root: Path) -> list[tuple[Path, Path]]:
    moves = []
    for source in root.iterdir():
        if not source.is_file() or source.name.startswith("."):
            continue
        category = source.suffix.lower().lstrip(".") or "no-extension"
        target_dir = root / category
        destination = target_dir / source.name
        if destination.exists():
            raise FileExistsError(destination)
        moves.append((source, destination))
    return moves
```

DONE WHEN Dry run | Collision safe | Temporary-directory tests | Backup/copy used for first real trial

CHAPTER 74

Project 3 — Accessible track library website

WHY THIS MATTERS This project joins semantic HTML, responsive CSS, JavaScript state, local storage and browser debugging.

Core understanding

Users add tracks, filter by title, calculate total duration and persist data locally. The interface works with keyboard and screen-reader announcements. No framework is required.

Acceptance: blank titles rejected; malformed stored data recovered; controls labelled; total updates; 200% zoom remains usable; user text is rendered through textContent.

Working method

131. Create index.html, styles.css, app.js and README.
132. Build semantic form and result list before styling.
133. Implement pure duration helpers and test them in the console or test runner.
134. Render items with createElement and textContent.
135. Persist a versioned JSON object to localStorage with validation on load.
136. Use browser DevTools breakpoints, console and accessibility tree.
137. Serve locally, test keyboard/zoom, then publish static files.

Worked example

JAVASCRIPT CORE

```
const STORAGE_KEY = "track-library:v1";

function saveTracks(tracks) {
 localStorage.setItem(STORAGE_KEY, JSON.stringify({ version: 1, tracks }));
}

function loadTracks() {
 try {
  const parsed = JSON.parse(localStorage.getItem(STORAGE_KEY) ?? "null");
  return parsed?.version === 1 && Array.isArray(parsed.tracks) ? parsed.tracks : [];
 } catch {
  return [];
 }
}
```

DONE WHEN Keyboard usable | Zoom usable | Storage validated | No unsafe HTML insertion

CHAPTER 75

Project 4 — C# API plus web client

WHY THIS MATTERS This capstone connects an ASP.NET Core API, SQLite, validation, tests and a browser client.

Core understanding

The API owns track data and exposes GET/POST endpoints. SQLite persists records. The web client fetches and displays them. Cross-origin configuration is restricted to the actual client origin in development.

Acceptance: duplicate track number returns conflict; invalid duration returns bad request; database operations are parameterized; API tests use an isolated database; secrets remain outside source.

Working method

138. Create API and test projects in Visual Studio or VS Code.
139. Design the schema and request/response contracts.
140. Implement repository methods with parameters and cancellation.
141. Map endpoints with validation and meaningful status codes.
142. Write integration tests before connecting the browser client.
143. Build the client and verify loading, empty and error states.
144. Run both through named tasks/launch profiles, inspect Git diff and publish separately.

Worked example

ASP.NET CORE ENDPOINT

```
app.MapPost("/api/tracks", async (CreateTrack request, TrackRepository repo, CancellationToken ct) =>
{
    if (string.IsNullOrWhiteSpace(request.Title) || request.Seconds <= 0)
        return Results.BadRequest(new { error = "title and positive seconds are required" });

    Track created = await repo.CreateAsync(request, ct);
    return Results.Created($"/api/tracks/{created.Id}", created);
});
```

DONE WHEN Contract tested | Database isolated | Client error state | Separate publish artifacts

CHAPTER 76

Your next 30 builds

WHY THIS MATTERS Skill comes from finishing increasingly independent projects, not from reading alone.

Core understanding

Repeat the safe loop with small projects that force one new concept at a time. Keep every finished project in Git with a README, tests appropriate to its risk and a short retrospective.

Suggested sequence: unit converter, expense calculator, text analyser, contact CLI, file searcher, CSV summarizer, REST client, SQLite tracker, accessible form, dashboard, authentication sample, background worker and deployable API.

Operational notes

- Build 1-5: console input, calculation, conditions, loops and functions.
- Build 6-10: files, JSON, modules, tests and Git branches.
- Build 11-15: SQLite, HTTP and async behaviour.
- Build 16-20: accessible browser UI and TypeScript.
- Build 21-25: APIs, authentication concepts, logging and deployment.
- Build 26-30: choose your own problem; write acceptance tests before implementation.

DONE WHEN Project finished | README verified | Tests pass | Retrospective written

PART APPENDICES

Fast Reference

Commands, shortcuts, syntax and fault-finding in one lookup section.

CHAPTER 77

Visual Studio Community shortcut sheet

WHY THIS MATTERS Live keyboard mappings can change; use command names as the reliable lookup key.

Core understanding

These are common Visual Studio defaults for the General development profile. A chord such as Ctrl+K, Ctrl+D means press the first combination, release it, then press the second.

Command	Shortcut	Purpose
New project	Ctrl+Shift+N	Create from a template
Open project/solution	Ctrl+Shift+O	Open project metadata
Save / Save all	Ctrl+S / Ctrl+Shift+S	Write current/all changes
Go to All	Ctrl+T or Ctrl+,	Search files, symbols and members
Feature Search	Ctrl+Q	Find IDE commands/settings
Quick Actions	Ctrl+.	Fix or refactor at caret
Format document	Ctrl+K, Ctrl+D	Format entire document
Build solution	Ctrl+Shift+B	Compile selected solution
Start / no debugger	F5 / Ctrl+F5	Run with/without debugger
Breakpoint	F9	Toggle line breakpoint
Step over / into / out	F10 / F11 / Shift+F11	Control paused execution
Stop debugging	Shift+F5	End debug session
Definition / references	F12 / Shift+F12	Navigate symbol
Solution Explorer	Ctrl+Alt+L	Open project tree
Output	Ctrl+Alt+O	Open detailed logs

CHAPTER 78

Visual Studio Code shortcut sheet

WHY THIS MATTERS Use Keyboard Shortcuts to inspect the installed binding and conflicts.

Core understanding

Mac is listed first because Keith uses macOS; Windows equivalents are beside it.

Action	macOS	Windows	Meaning
Command Palette	Shift+Command+P	Ctrl+Shift+P	Search every command
Quick Open	Command+P	Ctrl+P	Open file by name
Save	Command+S	Ctrl+S	Write active file
Settings	Command+,	Ctrl+,	Open settings
Explorer	Shift+Command+E	Ctrl+Shift+E	Project files
Search	Shift+Command+F	Ctrl+Shift+F	Workspace search
Source Control	Control+Shift+G	Ctrl+Shift+G	Git changes
Run and Debug	Shift+Command+D	Ctrl+Shift+D	Debug view
Extensions	Shift+Command+X	Ctrl+Shift+X	Manage extensions
Terminal	Control+`	Ctrl+`	Toggle terminal
Problems	Shift+Command+M	Ctrl+Shift+M	Diagnostics
Format document	Shift+Option+F	Shift+Alt+F	Run formatter
Quick Fix	Command+.	Ctrl+.	Code actions
Rename	F2	F2	Semantic rename
Start debug	F5	F5	Start/continue
Breakpoint	F9	F9	Toggle breakpoint

CHAPTER 79

Terminal command dictionary

WHY THIS MATTERS Commands act on the current shell and directory; verify both before using a destructive operation.

Intent	macOS/Linux shell	Windows PowerShell
Current directory	pwd	Get-Location
List files	ls -la	Get-ChildItem -Force
Change directory	cd path	Set-Location path
Create directory	mkdir name	New-Item -ItemType Directory name
Copy file	cp source target	Copy-Item source target
Move/rename	mv source target	Move-Item source target
Delete file	rm file	Remove-Item file
Find command	which command	Get-Command command
Environment variable	echo $NAME	$env:NAME
Clear screen	clear	Clear-Host

DESTRUCTIVE COMMANDS Never paste recursive delete, permission, package-install or remote-download commands without resolving the exact target and understanding every flag.

CHAPTER 80

Git command dictionary

WHY THIS MATTERS Use status and diff before every history-changing or publishing command.

Command	Meaning
git status	Show branch, working tree and staged state
git diff	Show unstaged changes
git diff --staged	Show proposed next commit
git switch -c name	Create and switch to a branch
git add path	Stage selected content
git commit -m message	Record staged snapshot locally
git fetch --prune	Download remote references and remove stale remote-tracking names
git pull	Fetch then integrate according to configuration
git push -u origin branch	Publish branch and set upstream
git log --oneline --graph	Inspect compact history graph
git restore path	Discard unstaged file changes; destructive
git revert commit	Create a new commit that reverses an earlier commit

CHAPTER 81

C# syntax sheet

WHY THIS MATTERS Use this as a lookup after understanding the foundational chapters.

C#

```
int count = 3;
decimal price = 12.50m;
string title = "Track";
bool locked = true;
int? optional = null;

if (locked) { Console.WriteLine(title); }
for (int i = 0; i < count; i++) { Console.WriteLine(i); }
foreach (string name in names) { Console.WriteLine(name); }

static int Add(int a, int b) => a + b;
var squares = numbers.Where(n => n > 0).Select(n => n * n).ToList();

try { await SaveAsync(ct); }
catch (IOException ex) { Console.Error.WriteLine(ex.Message); }
```

- Naming: PascalCase for public types/members, camelCase for locals/parameters.
- Enable nullable reference analysis.
- Dispose resources with using/await using.
- Prefer decimal for money and DateTimeOffset for instants.

CHAPTER 82

Python syntax sheet

WHY THIS MATTERS Python's readability depends on consistent indentation and explicit project environments.

PYTHON

```
count: int = 3
price: float = 12.50
title: str = "Track"
locked: bool = True

if locked:
    print(title)

for index, name in enumerate(names):
    print(index, name)

def add(a: int, b: int) -> int:
    return a + b

squares = [n * n for n in numbers if n > 0]

try:
    save()
except OSError as exc:
    print(exc)
```

- Use four spaces; never mix tabs and spaces.
- Use pathlib for paths and context managers for resources.
- Use a virtual environment per project.
- Type hints aid tools but require a checker for static enforcement.

CHAPTER 83

JavaScript and TypeScript syntax sheet

WHY THIS MATTERS Use const by default, validate external data and keep browser secrets out of source.

JAVASCRIPT

```
const count = 3;
let selected = null;

if (count > 0) {
  console.log(`Count: ${count}`);
}

for (const item of items) {
  console.log(item.name);
}

function add(a, b) { return a + b; }
const active = items.filter(item => item.active).map(item => item.name);

try {
  const result = await loadData();
} catch (error) {
  console.error(error);
}
```

TYPESCRIPT

```
type Track = { id: number; title: string; seconds: number };
function format(track: Track): string { return `${track.title}: ${track.seconds}`; }
```

CHAPTER 84

HTML and CSS syntax sheet

WHY THIS MATTERS Structure comes before styling; accessible meaning is part of the interface contract.

HTML

```
<main>
 <h1>Title</h1>
 <form>
  <label for="email">Email</label>
  <input id="email" name="email" type="email" required>
  <button type="submit">Save</button>
 </form>
</main>
```

CSS

```
* { box-sizing: border-box; }
body { font: 1rem/1.5 system-ui, sans-serif; }
.container { width: min(70rem, 100% - 2rem); margin-inline: auto; }
.grid { display: grid; gap: 1rem; grid-template-columns: repeat(auto-fit, minmax(15rem, 1fr)); }
:focus-visible { outline: 3px solid #2e74b5; outline-offset: 2px; }
```

CHAPTER 85

Troubleshooting decision tree

WHY THIS MATTERS Use the first failed layer to choose the next check.

Symptom	Check in order
Template missing	Product > workload/SDK > installer modification > supported target
Command missing	Correct product > folder/project open > extension/workload > trust/context > command search
Build fails	First error > project/configuration > restore > compiler detail in Output
Run fails	Build > startup target > runtime/SDK > working directory > configuration/environment
Breakpoint unbound	Debug session > source built > symbols match > correct process/module > optimization
Import/package missing	Selected interpreter/runtime > active environment > manifest > restore/install > working directory
Git view empty	Repository root > Git installed > .git present > output log > trust/permissions
Tests not found	Project root > adapter/extension > test framework > discovery settings > output
Works locally only	Clean clone > versions > configuration > paths/case > network/database > build artifact

CHAPTER 86

Glossary

WHY THIS MATTERS Precise vocabulary makes technical instructions testable.

Term	Meaning
API	A defined interface through which software components communicate.
Build	The process that transforms source and configuration into outputs.
CLI	Command-line interface.
Commit	A Git snapshot with identity, parents and message.
Compiler	A tool that translates source into another representation and reports invalid code.
Debugger	A tool for pausing execution and inspecting state.
Dependency	External code or another project required by the current project.
IDE	Integrated development environment combining editor, project, build, debug and other tools.
Interpreter	A runtime that executes source or an intermediate form.
Library	Reusable code consumed by another program.
Package	A versioned distribution of code and metadata.
Project	A buildable logical unit plus source and settings.
Repository	A version-controlled work tree and its history.
Runtime	Software environment that executes a program.
SDK	Software development kit: compilers, libraries and tools for a target.
Shell	A command interpreter such as zsh, bash or PowerShell.
Solution	A Visual Studio container for one or more projects.
Test	Executable verification that observed behaviour meets an expectation.
Workspace	The folder or multi-folder context open in an editor.

CHAPTER 87

Official verification sources

WHY THIS MATTERS Software changes; these primary sources override a printed command when versions differ.

Core understanding

Visual Studio IDE documentation: https://learn.microsoft.com/en-us/visualstudio/ide/

Visual Studio 2026 release notes: https://learn.microsoft.com/en-us/visualstudio/releases/2026/release-notes

Visual Studio downloads and licensing: https://visualstudio.microsoft.com/

Visual Studio Code documentation: https://code.visualstudio.com/docs

Visual Studio Code 1.130 release notes: https://code.visualstudio.com/updates/v1_130

.NET and C# documentation: https://learn.microsoft.com/dotnet/

C# language reference: https://learn.microsoft.com/dotnet/csharp/language-reference/

Python in VS Code: https://code.visualstudio.com/docs/python/python-tutorial

JavaScript in VS Code: https://code.visualstudio.com/docs/languages/javascript

TypeScript in VS Code: https://code.visualstudio.com/docs/languages/typescript

Git documentation: https://git-scm.com/doc

MDN Web Docs for web standards: https://developer.mozilla.org/

Operational notes

- Primary product documentation was checked on 25 August 2026.
- Package and framework examples are conceptual; verify current supported versions before installation.

DONE WHEN Live docs checked for version-sensitive task | Installed command/menu verified

CLOSING DIRECTIVE

Build small. Read the evidence. Protect the history. Finish the project.

The goal is not to remember every command. The goal is to know what you are trying to prove, which tool exposes the evidence, and how to make the next change safely.