VS CODE

Essentials

Getting Started · Command Palette & Quick Open · Editing Efficiently
Extensions · Workspaces & Settings · Terminal & Tasks
Git Integration · Debugging · Snippets & Keybindings · Remote Development

Philip Osztromok

Generated with Claude

Table of Contents

VS Code Essentials — 10 Chapters

🚀 Getting Started

Welcome to VS Code Essentials — a course about the editor itself, not any one language you might write in it. This chapter covers installing VS Code, what you'll see on first launch, and a tour of the five main regions of the interface you'll be living in for the rest of this course.

Installing VS Code

VS Code is downloaded from code.visualstudio.com, with installers for Windows, macOS, and Linux. It's free, and its core (the "Code - OSS" project) is open source under the MIT license — the official Microsoft-branded download adds a few proprietary extras (like telemetry and the Marketplace connection) on top of that open core.

On most systems, a package manager is faster than the website:

```
# Windows (winget)
winget install Microsoft.VisualStudioCode

# macOS (Homebrew)
brew install --cask visual-studio-code

# Debian/Ubuntu (apt, after adding Microsoft's repository)
sudo apt install code
```

⚠ VS Code ≠ Visual Studio

These are two entirely different products that happen to share a name and a publisher. **Visual Studio** is Microsoft's large, long-standing full IDE, historically centered on C#/.NET and C++, Windows-only until relatively recently. **VS Code** is a lightweight, cross-platform, extension-driven code editor, language-agnostic by design. If a tutorial says "open Visual Studio" and you open VS Code (or vice versa), nothing will make sense — always double-check which one is actually meant.

🎬 First Launch — What You'll See

On first launch, VS Code opens to a **Welcome** tab — recent-file shortcuts, a "Open Folder" button, and links to customize your theme and install extensions. This is a normal editor tab, not a separate wizard — you can close it like any other tab, and reopen it later from **Help → Welcome** if you want it back.

🗺️ The UI Tour

Five regions make up almost everything you'll interact with:

Activity Bar

The far-left strip of icons — Explorer, Search, Source Control, Run & Debug, Extensions. Clicking one changes what the Side Bar shows.

Side Bar

The panel next to the Activity Bar — its content is context-sensitive, following whichever Activity Bar icon is selected.

Editor Groups

The main area where files are actually open — can be split into multiple groups, side by side or stacked.

Panel

The bottom area — Terminal, Output, Debug Console, and Problems all live here as tabs.

Status Bar

The thin strip along the very bottom — git branch, language mode, line/column position, and error/warning counts.

The **Explorer** (the first Activity Bar icon, usually selected by default) shows your currently open folder's file tree in the Side Bar — this is the view you'll spend the most time in. The **Source Control** icon shows a small badge with a number whenever there are uncommitted git changes, a useful at-a-glance signal even before opening it (Chapter 7 covers this in depth).

📂 Opening a Folder vs. a File

VS Code is built around **projects**, not individual files. **File → Open Folder** (not just "Open File") is the normal way to start working — once a folder is open, the Explorer shows its full tree, search (Chapter 3) covers every file in it, and git integration (Chapter 7) picks up its repository automatically. Opening a single loose file works too, but you lose all of that project-wide context.

Where VS Code Sits: Plain Editor vs. Full IDE

Type	Examples	Characteristic
Plain text editor	Notepad, gedit	Opens files, edits text — no project awareness, no language intelligence
VS Code	—	Project-aware, with language intelligence (autocomplete, error-checking, debugging) added *through extensions* rather than built in for every language at once
Full IDE	Visual Studio, a JetBrains IDE (IntelliJ, PyCharm…)	Deep, language-specific tooling built directly into the product, heavier to install and run

This "extension-driven middle ground" is VS Code's whole identity — a fast, general-purpose core, made as deep as any single project needs through the extensions covered in Chapter 4.

💻 Coding Challenges

Exercise 1: Install & Identify

Install VS Code, launch it, and open any folder on your machine via **File → Open Folder**. Identify all five UI regions from this chapter (Activity Bar, Side Bar, Editor Groups, Panel, Status Bar) and note one specific thing you see in each.

→ Solution

Exercise 2: Split the Editor

Open two different files from your folder side by side in two Editor Groups, then open and close the integrated terminal (Panel) at least once.

→ Solution

Exercise 3: Status Bar Investigation

With a file open, click its language-mode indicator in the Status Bar and note what options appear. Then click the line/column indicator and use it to jump to a specific line number.

→ Solution

🎯 What's Next

Next chapter: **The Command Palette & Quick Open** — `Ctrl+Shift+P`, `Ctrl+P`, and keyboard-first navigation.

⌨️ The Command Palette & Quick Open

Chapter 1 toured the interface by sight. This chapter covers the single most important habit for using VS Code efficiently: reaching for the keyboard, not the mouse, via two related but distinct tools — the Command Palette and Quick Open.

🎛️ The Command Palette (Ctrl+Shift+P)

`Ctrl+Shift+P` (`Cmd+Shift+P` on macOS) opens the **Command Palette** — a searchable list of *every single command* VS Code can run, fuzzy-matched as you type. Instead of hunting through menus for "change the color theme," you press `Ctrl+Shift+P`, type `theme`, and **Preferences: Color Theme** appears near the top of the results.

```
// A few genuinely useful commands worth trying right now:
Format Document
Toggle Word Wrap
Preferences: Color Theme
Preferences: Open Settings (JSON)
View: Toggle Terminal
```

Fuzzy matching means you don't need the exact wording — typing `fmtdoc` will still find **Format Document**, since fuzzy search matches the letters in order, not as a contiguous substring.

Quick Open (Ctrl+P)

`Ctrl+P` opens **Quick Open** — a fuzzy search over every file in your currently open folder (Chapter 1). Typing a few letters of a filename jumps straight to it, no clicking through the Explorer's folder tree required.

⚠ Ctrl+P and Ctrl+Shift+P Do Genuinely Different Things

These two shortcuts look almost identical and are easy to confuse early on: `Ctrl+P` (Quick Open) searches for **files** to open; `Ctrl+Shift+P` (Command Palette) searches for **commands** to run. Typing a filename into the Command Palette, or a command name into Quick Open, won't find what you're looking for — the extra `Shift` is the whole difference between "open a file" and "do something."

🔗 Combining Them — Prefix Characters in Quick Open

Quick Open (`Ctrl+P`) isn't just for files — typing a special character first switches its mode entirely:

Prefix	What it searches
`>`	Commands — typing `>` in Quick Open actually opens the same list as `Ctrl+Shift+P`. They're the same underlying box.
`@`	Symbols in the *current file* — functions, classes, variables, jump directly to a definition.
`#`	Symbols across the *entire open folder* — find a function by name without knowing which file it's in.
`:`	A line number in the current file — the same line-jump from Chapter 1's Status Bar exercise, reachable from Quick Open too.

Realizing `Ctrl+P` then `>` gives you the Command Palette explains why they feel so closely related — they genuinely share one search box, just entered from two different shortcuts and two different default modes.

🎯 Why Keyboard-First Matters

The value isn't just raw speed (though it is faster) — it's **discoverability without memorization**. You don't need to know that word wrap lives under **View** in the menu bar; you only need to guess a plausible word ("wrap") and let fuzzy search do the rest. This scales far better than memorizing menu locations as VS Code's feature set (and your own installed extensions, Chapter 4) grows.

A Familiar Idea, If You've Used Vim

If you've worked through the site's own Vim course, the Command Palette should feel conceptually familiar: Vim's `:` command-line mode is a single, searchable entry point for running any command, rather than a menu you navigate visually. VS Code's Command Palette is the same underlying idea — one search box, near-universal reach — expressed in a GUI editor instead of a terminal one.

🧠 Building Muscle Memory

A handful of shortcuts are worth committing to memory early, since they'll be used constantly for the rest of this course:

Ctrl+P

Quick Open — jump to any file by name.

Ctrl+Shift+P

Command Palette — run any command by name.

Ctrl+`

Toggle the integrated terminal (from Chapter 1).

Ctrl+B

Toggle the Side Bar — reclaim screen space when you don't need the Explorer visible.

💻 Coding Challenges

Exercise 1: Change Your Theme by Command

Using only `Ctrl+Shift+P`, find and run the command to change your color theme, and pick a different one than the default. Do not use the mouse to navigate any menu.

→ Solution

Exercise 2: Jump Without the Explorer

With a folder open containing at least 3 files, use `Ctrl+P` to jump directly to a specific file by typing only part of its name — without ever clicking anything in the Explorer.

→ Solution

Exercise 3: Try Every Prefix

Open Quick Open (`Ctrl+P`) and try all four prefix characters from this chapter's table (`>`, `@`, `#`, `:`) one at a time, noting what each one searches.

→ Solution

🎯 What's Next

Next chapter: **Editing Efficiently** — multi-cursor, multi-select, column selection, and search & replace.

✂️ Editing Efficiently

Chapter 2 covered navigating faster. This chapter covers editing faster — multi-cursor and multi-select techniques that turn a repetitive, many-step edit into a single action, plus workspace-wide search & replace, including regex mode.

🖱️ Multi-Cursor Editing

`Alt+Click` (`Option+Click` on macOS) anywhere in the editor adds a **new cursor** at that exact position — every cursor you place accepts typing simultaneously, so one keystroke edits every location at once.

```
// Alt+Click before each "TODO" below places 3 cursors at once —
// typing "FIXME" now replaces all three simultaneously:
// TODO: refactor this
// TODO: add tests
// TODO: update docs
```

`Ctrl+Alt+Down` / `Ctrl+Alt+Up` (`Cmd+Option+Down`/`Up` on macOS) adds a cursor directly on the line below/above the current one — useful for editing several consecutive lines identically, without needing to click each one precisely.

⚠ Multiple Cursors Persist Until You Collapse Them

After a multi-cursor edit, the extra cursors **stay active** — the next thing you type still goes to all of them, not just one. Pressing `Escape` collapses back to a single cursor at your last edit position. Forgetting this is a common way to accidentally type the same text into several unrelated places right after finishing a multi-cursor edit.

🎯 Multi-Select by Match (Ctrl+D)

`Ctrl+D` (`Cmd+D` on macOS) selects the word under your cursor, then — pressed again — selects the *next occurrence* of that same word, adding a new cursor there too. Keep pressing `Ctrl+D` to keep adding matches one at a time, editing all of them together once you type.

```
// with the cursor on "color" below, Ctrl+D twice selects both
// occurrences at once — typing "colour" fixes both simultaneously:
let favoriteColor = "blue";
let secondaryColor = "green";
```

`Ctrl+Shift+L` (`Cmd+Shift+L` on macOS) skips the "press repeatedly" step entirely — it selects *every* occurrence of the currently selected word in the file at once, in a single keystroke.

▭ Column (Box) Selection

Holding `Shift+Alt` while dragging with the mouse selects a **rectangular block** of text spanning multiple lines, rather than a normal line-by-line selection. This is the tool for editing tabular or column-aligned text — adding the same prefix to the start of several lines, for example, without needing multi-cursor placement on each one individually.

A Familiar Idea, Again From Vim

If you've used the site's Vim course, this maps directly onto Vim's **block-visual mode** (`Ctrl+V` in Vim) — select a rectangular region spanning multiple lines, then insert or delete across all of them at once. Different keystroke, same underlying idea: sometimes the shape you want to select is a column, not a line.

🔍 Search & Replace Across Files

`Ctrl+Shift+F` (`Cmd+Shift+F` on macOS) opens workspace-wide search — searching (and optionally replacing) across *every file* in your open folder at once, not just the currently open one. Results are grouped by file, each showing the matching line in context; clicking a result jumps straight to it.

⚠ Preview Before You Replace All

Workspace-wide "Replace All" changes every match across every file in one action — always expand and review the results first (each result is individually toggleable, letting you exclude specific matches) rather than trusting a search term to have matched only what you intended. This matters even more once regex mode (below) is involved, since a slightly-too-broad pattern can match far more than expected.

🧩 Regex Mode in Search

The search box has a small `.*` icon — toggling it switches from plain-text search to **regular expression** search, using the exact regex syntax covered in the site's own *Learning Regular Expressions* course (`regex1`). Capture groups from the pattern are available in the replace field as `$1`, `$2`, and so on.

```
// Search (regex mode on):  console\.log\((.*)\)
// Replace:                 logger.debug($1)
// Turns every console.log(x) into logger.debug(x), keeping
// whatever was originally inside the parentheses.
```

Alt+Click

Adds a cursor at any position; type once, edit everywhere at once.

Ctrl+D / Ctrl+Shift+L

Select the next (or every) occurrence of a word for simultaneous editing.

Shift+Alt+drag

Column/box selection — for tabular or column-aligned text.

Ctrl+Shift+F

Workspace-wide search & replace, with an optional regex mode.

💻 Coding Challenges

Exercise 1: Multi-Cursor Cleanup

Open any file with at least 3 similar lines (or create one). Use `Ctrl+Alt+Down` to place cursors on all of them, then type the same addition on every line at once.

→ Solution

Exercise 2: Rename Every Occurrence

In a file containing the same word 3+ times, use `Ctrl+Shift+L` to select every occurrence at once and rename all of them with a single edit.

→ Solution

Exercise 3: A Workspace-Wide Regex Replace

Using `Ctrl+Shift+F` with regex mode enabled, search your open folder for a simple pattern with a capture group, and use `$1` in the replace field — reviewing the results before confirming, per this chapter's own warn-box.

→ Solution

🎯 What's Next

Next chapter: **Extensions** — the Marketplace, evaluating an extension before installing it, and a few genuinely essential picks.

🧩 Extensions

Chapter 1's comparison-box named extensions as the mechanism that lets VS Code stay lightweight by default while going as deep as any single project needs. This chapter covers that mechanism directly: the Marketplace itself, how to actually evaluate an extension before installing it, a few genuinely essential picks, and why more extensions isn't automatically better.

🛒 The Extensions View & Marketplace

`Ctrl+Shift+X` (`Cmd+Shift+X` on macOS) opens the **Extensions** view in the Side Bar — or click the Extensions icon in the Activity Bar from Chapter 1. Searching here queries the **VS Code Marketplace**, Microsoft's official public extension registry; clicking **Install** on a result downloads and activates it immediately, no restart required for most extensions.

🔎 Evaluating an Extension Before Installing

Anyone can publish to the Marketplace — before installing, a few quick checks are worth the extra 30 seconds:

Publisher

A blue checkmark next to the publisher name means Microsoft has verified their identity — not a guarantee of quality, but a real signal against outright impersonation.

Install count & rating

A widely-installed, well-rated extension has been battle-tested by far more people than a brand-new one with a handful of installs.

Last updated

An extension untouched for years may be abandoned — a real problem if VS Code's own APIs change underneath it.

Repository link

Most trustworthy extensions link an open-source repository — a closed-source extension asking for broad permissions deserves extra scrutiny.

⭐ A Few Genuinely Essential Picks

Three extensions come up constantly regardless of what language you're working in:

- **ESLint** — runs a linter against your JavaScript/TypeScript as you type, flagging likely bugs and style issues directly in the editor, before you'd ever see them at runtime.
- **Prettier** — an opinionated, zero-configuration code formatter. Rather than debating formatting style, Prettier just picks one and applies it consistently on save — genuinely reduces a whole category of pointless team disagreement.
- **GitLens** — supercharges the built-in Git integration (Chapter 7) with inline "blame" annotations (who last changed this line, and when), a rich commit history view, and easy comparison between branches or commits.

⚖️ Extension Bloat — Why More Isn't Better

Every installed extension is code that runs continuously in the background — more installed extensions means slower startup, more memory used, and more surface area for something to conflict or break. Before installing, it's worth asking whether you'll actually use an extension regularly, or whether it's solving a problem you don't currently have. A handful of well-chosen extensions genuinely used every day beats a large pile installed "just in case."

⚠ An Extension Runs Arbitrary Code With Your Permissions

An installed extension isn't sandboxed the way a browser tab is — it runs as ordinary code with the same file-system and network access you have. This is a genuine supply-chain risk: malicious or compromised extensions have appeared on the Marketplace in the past, sometimes impersonating popular ones with a near-identical name. Treat installing an extension with the same caution as installing a package via `pip` or `npm` — check the publisher, the install count, and the repository link from this chapter's evaluation list before trusting it with your machine.

Closing the Loop on Chapter 1's Comparison

Chapter 1 placed VS Code between a plain text editor and a full IDE, calling out that its depth comes "through extensions rather than built in for every language at once." This chapter is that mechanism made concrete: a full IDE ships its language tooling as one large, vetted, pre-integrated package; VS Code instead lets you assemble exactly the tooling you need, extension by extension — more flexible, but shifting the evaluation and trust decisions onto you as the person choosing what to install.

💻 Coding Challenges

Exercise 1: Evaluate Before Installing

Search the Marketplace for any extension related to a language or tool you use. Before installing it, check its publisher verification, install count, rating, and last-updated date — write down what you found for each.

→ Solution

Exercise 2: Install Prettier & Format a File

Install the Prettier extension, then open any JavaScript, JSON, or CSS file and use the Command Palette (Chapter 2) to run "Format Document" on it.

→ Solution

Exercise 3: Audit Your Installed Extensions

Open the Extensions view and list every extension you currently have installed. For each one, decide honestly whether you've actually used it in the last month — and uninstall any that you haven't, per this chapter's bloat discussion.

→ Solution

🎯 What's Next

Next chapter: **Workspaces & Settings** — User vs. Workspace settings, `settings.json`, and recommended extensions via `extensions.json`.

⚙️ Workspaces & Settings

Every setting you change in VS Code has to live somewhere — this chapter covers exactly where, the difference between a setting that follows you everywhere and one scoped to a single project, and how to work across several unrelated folders at once with a multi-root workspace.

👤 User Settings vs. Workspace Settings

VS Code has two layers of settings, checked in order:

Layer	Scope	Stored in
User settings	Every project, everywhere on this machine	A single global `settings.json`
Workspace settings	Only the currently open folder	`.vscode/settings.json`, inside the project itself

Where both are set for the same setting, **Workspace settings win** — this is what lets one project enforce 2-space indentation while your personal default everywhere else is 4, without those two preferences ever conflicting.

📝 Editing settings.json

`Ctrl+,` (`Cmd+,` on macOS) opens the **Settings UI** — a searchable, friendlier view with tabs for User and Workspace. For settings not exposed there, or to paste in a configuration directly, the Command Palette's (Chapter 2) **Preferences: Open Settings (JSON)** command opens the raw file:

```
// settings.json
{
 "editor.formatOnSave": true,
 "editor.tabSize": 2,
 "files.autoSave": "afterDelay",
 "editor.fontSize": 14
}
```

📋 Common Settings Worth Knowing

editor.formatOnSave

Runs your formatter (Chapter 4's Prettier, for example) automatically every save.

editor.tabSize

How many spaces one indentation level represents.

files.autoSave

Saves automatically after a delay or on focus change, instead of only on `Ctrl+S`.

editor.fontSize

The editor's font size, independent of your OS-wide display settings.

🗂️ Multi-Root Workspaces

Normally, "Open Folder" gives you one project. A **multi-root workspace** combines several unrelated folders into one window — useful when a frontend repo and a backend repo need to be visible side by side, for instance. **File → Add Folder to Workspace** adds a second root; **File → Save Workspace As** saves the combination as a `.code-workspace` file you can reopen later, with each root folder's own Workspace settings kept separate.

📌 Recommended Extensions via extensions.json

Chapter 4 covered choosing extensions for yourself. A project can also declare its own recommendations, in `.vscode/extensions.json`:

```
// .vscode/extensions.json
{
 "recommendations": [
  "dbaeumer.vscode-eslint",
  "esbenp.prettier-vscode"
 ]
}
```

When someone opens this folder for the first time, VS Code shows a prompt offering to install exactly these extensions — a lightweight, version-controlled way to keep a team aligned on the essentials (Chapter 4's ESLint/Prettier picks are a natural fit here) without forcing anyone's personal, unrelated extension choices on the whole project.

⚠ Commit Shared Standards, Not Personal Preferences

`.vscode/settings.json` and `.vscode/extensions.json` are typically committed to the repository, so the whole team gets the same project-level standards automatically. Be deliberate about what goes in there: tab size, format-on-save, and required linting extensions are genuinely team-wide concerns. A specific color theme, personal font size, or your own keybindings are not — those belong in your *User* settings, kept off the shared file entirely.

A Familiar Split: Global vs. Project-Local

If you've worked through the site's Python courses, this User-vs-Workspace split mirrors a familiar idea from a different angle: a Python `venv` (Course 1, Chapter 9) keeps one project's dependencies isolated from your machine's global Python install, the same way Workspace settings keep one project's preferences isolated from your machine-wide User settings — in both cases, project-local state overriding a broader global default, without the two ever colliding.

💻 Coding Challenges

Exercise 1: Set a Workspace-Only Preference

In one project, set `editor.tabSize` to 2 as a *Workspace* setting (not User). Confirm a different project still uses your User-level default by opening it and checking the same setting.

→ Solution

Exercise 2: Create an extensions.json

In a project's `.vscode` folder, create an `extensions.json` recommending 2 extensions from Chapter 4. Close and reopen the folder to see VS Code's install prompt appear.

→ Solution

Exercise 3: Build a Multi-Root Workspace

Open one folder, then use **File → Add Folder to Workspace** to add a second, unrelated folder. Save the result with **File → Save Workspace As**, then close and reopen it from the saved `.code-workspace` file.

→ Solution

🎯 What's Next

Next chapter: **The Integrated Terminal & Tasks** — running commands without leaving the editor, and `tasks.json` for build/watch/test tasks.

🖥️ The Integrated Terminal & Tasks

Chapter 1 introduced toggling the terminal panel with `Ctrl+`` . This chapter goes deeper: running multiple terminals at once, choosing which shell they use, and — the real payoff — `tasks.json`, which turns a command you'd otherwise retype constantly into a one-click, keybindable action.

🖱️ The Integrated Terminal, In Depth

`Ctrl+`` toggles the terminal panel open and closed, but clicking the + icon in the terminal panel opens a *new, additional* terminal — several can run simultaneously (one running a dev server, another free for git commands, for instance), switchable via the dropdown next to the +.

Unlike opening a standalone terminal application, VS Code's integrated terminal starts *already inside your project's folder* — no manual `cd` required, since it inherits the working directory from whatever's currently open (Chapter 1).

🐚 Terminal Profiles — Choosing Your Shell

VS Code doesn't lock you into one shell. On Windows, it can launch PowerShell, Command Prompt, Git Bash, or a WSL shell; on macOS/Linux, typically `zsh` or `bash`. The Command Palette's (Chapter 2) **Terminal: Select Default Profile** command lets you choose which one opens by default — useful since a tutorial or team's shared shell scripts might assume a specific shell (Bash-style syntax, for instance) that differs from your system's own default.

▭ Splitting Terminals

`Ctrl+Shift+5` (or the split icon in the terminal panel) splits the current terminal into two side-by-side panes — useful for watching a running process's output in one pane while typing commands in the other, without switching back and forth between separate terminal tabs.

⚙️ tasks.json — Automating Repeated Commands

Running the same command over and over — a build step, a test suite — gets old fast. **Terminal → Configure Tasks** creates `.vscode/tasks.json`, where a task is just a labeled shell command:

```
// .vscode/tasks.json
{
 "version": "2.0.0",
 "tasks": [
  {
   "label": "Build",
   "type": "shell",
   "command": "npm run build"
  }
 ]
}
```

Running **Tasks: Run Task** from the Command Palette now shows **Build** as an option — select it, and VS Code runs the command in a new terminal automatically, no retyping needed.

🔨 A Worked Example — Build & Watch Tasks

```
{
 "version": "2.0.0",
 "tasks": [
  {
   "label": "Build",
   "type": "shell",
   "command": "npm run build",
   "problemMatcher": ["$tsc"]
  },
  {
   "label": "Watch",
   "type": "shell",
   "command": "npm run watch",
   "isBackground": true,
   "problemMatcher": ["$tsc-watch"]
  }
 ]
}
```

`"isBackground": true` marks **Watch** as a long-running task that doesn't "finish" the way **Build** does — appropriate for a file-watcher that keeps running until stopped. `problemMatcher` is a genuinely deep feature on its own (parsing a tool's raw output to populate the editor's Problems panel with clickable errors) — worth knowing it exists and that VS Code ships built-in matchers like `$tsc` for common tools, without needing to master writing a custom one yet.

⚠ Background Tasks Keep Running Until You Stop Them

An `isBackground` task (or any long-running command started in a terminal) doesn't stop when you switch away from its terminal tab — it keeps consuming CPU, memory, or a network port in the background, invisibly, until explicitly killed. Leaving several forgotten watch processes running across piled-up terminal tabs is a common, easy-to-miss source of a sluggish machine. Check the terminal dropdown from this chapter's first section periodically, and close (trash-can icon) any terminal whose process you no longer need.

tasks.json vs. package.json Scripts

If you've used Node.js, a `package.json` `"scripts"` section already does something similar — `npm run build`. The difference: `tasks.json` isn't limited to npm scripts at all (it can wrap any shell command, in any language's ecosystem), and it integrates directly with the Problems panel and keybindings inside VS Code. `package.json` scripts are editor-agnostic and work identically for any teammate regardless of what tool they use — the two aren't competitors so much as different layers: a project might reasonably have both, with a `tasks.json` task that simply runs an existing `npm` script.

Multiple terminals

Click + to open another; switch between them via the dropdown.

Terminal profiles

Choose which shell (PowerShell, Bash, WSL…) opens by default.

tasks.json

A labeled shell command, run via the Command Palette without retyping it.

isBackground / problemMatcher

Marks a long-running task; parses its output into the Problems panel.

💻 Coding Challenges

Exercise 1: Two Terminals, One Panel

Open a terminal, then click + to open a second one. Run a different simple command in each (e.g. `echo hello` in one, `dir`/`ls` in the other), and switch between them using the dropdown.

→ Solution

Exercise 2: Your First Task

In any project, create `.vscode/tasks.json` with one task that runs a simple command (like `echo Build complete`). Run it via the Command Palette's "Tasks: Run Task" instead of typing the command directly.

→ Solution

Exercise 3: Split and Compare

Open a terminal and split it with `Ctrl+Shift+5`. In one pane, run a command that produces ongoing output (like a ping, or a long-running script); in the other, run ordinary commands while the first keeps going.

→ Solution

🎯 What's Next

Next chapter: **Git Integration** — the Source Control panel, staging/committing/pushing, and resolving merge conflicts visually.

🔀 Git Integration

The site's own Git & GitHub courses (`git1-git3`) taught git from the command line — staging, committing, branching, merging. This chapter covers the exact same operations through VS Code's visual **Source Control** panel: same underlying git repository, no new tool, just a different way to drive it.

📋 The Source Control Panel

`Ctrl+Shift+G` (`Cmd+Shift+G` on macOS), or the branch icon in the Activity Bar (Chapter 1 — the one with the small number badge whenever there are uncommitted changes), opens the **Source Control** panel. It lists every change in your working directory, grouped into **Changes** (modified or untracked files) and **Staged Changes**.

➕ Staging & Committing

Hovering a file under **Changes** reveals a `+` icon — clicking it stages that file, the exact equivalent of `git add <file>`. The `+` on the **Changes** header itself stages everything at once (`git add .`). Type a commit message into the box at the top of the panel, then click the checkmark icon — the same commit that `git commit -m "..."` would produce.

👁️ Viewing Diffs Inline

Clicking any changed file in the Source Control panel opens a **diff view** automatically — the old and new versions side by side (or inline, depending on your settings), with additions highlighted green and removals highlighted red. No need to run `git diff` manually and read raw `+`/`-` output in a terminal — this is that same information, rendered visually.

☁ Pushing, Pulling, and Syncing

The **Sync Changes** button (or the `...` menu at the top of the panel) exposes **Push**, **Pull**, and a combined **Sync** — pull then push in one action. The Status Bar (Chapter 1) shows a small icon with two numbers whenever your branch is ahead/behind its remote, updating live as you commit.

🌿 Branches

The current branch name appears in the Status Bar's bottom-left corner — clicking it opens a quick-pick list of every local and remote branch, letting you switch or create a new one without recalling `git checkout -b <name>` syntax from memory.

🧩 Resolving Merge Conflicts Visually

When a merge produces a conflict, VS Code annotates the file directly in the editor with clickable inline actions above each conflicting block: **Accept Current Change**, **Accept Incoming Change**, **Accept Both Changes**, and **Compare Changes** (a side-by-side view of exactly what differs). This is a genuine quality-of-life upgrade over manually editing raw `<<<<<<<`/`=======`/`>>>>>>>` conflict markers by hand in a plain text editor.

Same Git, Different Steering Wheel

Every action in this chapter produces the *exact same* git repository state as the equivalent command from `git1`-`git3` — staging via `+` is `git add`, the checkmark is `git commit`, nothing here is a VS-Code-specific format or a parallel system. Knowing what each button actually does underneath (because you've already run the command-line version) is what makes trusting the visual shortcut reasonable — it's convenience layered on the same tool, not a replacement for understanding it.

⚠ The Visual Tool Doesn't Replace Understanding Git

When something goes genuinely wrong — a bad merge, a commit that needs undoing, history that needs inspecting in a way the panel doesn't expose — the command-line skills from `git1`-`git3` are still the real fallback. The integrated terminal from Chapter 6 is right there for exactly this: nothing stops you from running `git log`, `git reset`, or any other command directly, in the same window, the moment the visual panel's simplified view isn't enough.

Source Control panel

`Ctrl+Shift+G` — Changes and Staged Changes, grouped and clickable.

Stage & commit

`+` to stage, a message box and checkmark to commit.

Inline diff view

Green/red color-coded changes, no manual `git diff` needed.

Visual conflict resolution

Accept Current/Incoming/Both, right inside the editor.

💻 Coding Challenges

Exercise 1: Stage and Commit Visually

In a git repository, make a small change to a file. Stage it using the `+` icon in the Source Control panel, write a commit message, and commit it using the checkmark — without typing a single git command.

→ Solution

Exercise 2: Read a Diff

Modify a file (change a line, add a new one), then click it in the Source Control panel before staging it. Identify which lines are shown in green vs. red in the diff view.

→ Solution

Exercise 3: Switch Branches from the Status Bar

Click the branch name in the Status Bar and use the quick-pick list to create a new branch. Confirm the Status Bar updates to show the new branch name.

→ Solution

🎯 What's Next

Next chapter: **Debugging** — `launch.json`, breakpoints, watch expressions, and the call stack.

🐞 Debugging

Most beginners debug almost entirely with `print()` or `console.log()` statements — a real, common habit, and not a wrong one. But VS Code has a genuinely powerful visual debugger built in, often underused simply because it's less immediately obvious than adding a print statement. This chapter covers it end to end: configuring how your code runs, setting breakpoints, inspecting state, and stepping through execution.

▶️ The Run & Debug View

The **Run & Debug** icon in the Activity Bar (Chapter 1 — the play button with a bug) opens the debug panel. `F5` starts a debug session; for many languages, VS Code can auto-detect how to run a simple script the first time, prompting you to create a proper configuration if it needs more information.

⚙️ launch.json — Configuring How to Run Your Code

For anything beyond a single trivial script, `.vscode/launch.json` tells the debugger exactly how to start your program:

```
// .vscode/launch.json
{
 "version": "0.2.0",
 "configurations": [
  {
   "name": "Launch Current File",
   "type": "python",
   "request": "launch",
   "program": "${file}"
  }
 ]
}
```

`type` selects which debugger extension handles the session (installed as part of that language's extension, Chapter 4); `request: "launch"` means VS Code starts the program itself (as opposed to `"attach"`, connecting to something already running); `${file}` is a built-in variable meaning "whichever file is currently open."

🔴 Breakpoints

Clicking in the gutter — the empty space just left of a line number — places a red dot: a **breakpoint**. When the debugger reaches that line, execution *pauses* right before running it, with the program's entire current state available to inspect. Right-clicking a breakpoint offers **Edit Breakpoint**, letting you add a **condition** (e.g. only pause when `i > 10`) — useful inside a loop where you only care about one specific iteration, not every single pass.

🔬 Inspecting State When Paused

While paused, the debug panel's **Variables** section automatically lists every variable in scope — local and global — with their current values, no setup required. Hovering your mouse over any variable name directly in the editor shows its value in a small tooltip, right where you're already looking. The **Watch** section lets you add a specific expression (not just a bare variable — `len(items) > 0` works too) that's continuously re-evaluated and displayed every time execution pauses.

📚 The Call Stack

The **Call Stack** panel shows the full chain of function calls that led to the current paused line — if `main()` called `process_order()` which called `validate()`, all three appear, in order. Clicking any frame in that list jumps the editor to that exact point, letting you inspect variables as they existed at each level of the call chain — genuinely useful for understanding *how* execution arrived somewhere, not just where it currently is.

👣 Stepping Through Code

Step Over (F10)

Runs the current line and moves to the next one — if it's a function call, runs the whole function without stepping into it.

Step Into (F11)

If the current line calls a function, jumps *inside* it, pausing at its first line.

Step Out (Shift+F11)

Finishes running the current function and pauses back where it was called from.

Continue (F5)

Resumes normal execution until the next breakpoint (or the program ends).

🔧 A Worked Example

```
def average(numbers):
    total = 0
for n in numbers:
        total += n
    return total / len(numbers) - 1 # bug: the -1 shouldn't be here

result = average([10, 20, 30])
print(result)   # 19.0 — expected 20.0, something's wrong
```

Set a breakpoint on the `return` line, press `F5`. Execution pauses right before returning — the Variables panel shows `total = 60`, `numbers = [10, 20, 30]`. Hovering over `total / len(numbers)` in the editor (or adding it to Watch) shows `20.0` — the correct average, before the stray `- 1` is applied. The bug is now visible directly, without ever adding a single `print()` line to the function.

The Debugger vs. print() Debugging

Print debugging requires guessing in advance what you'll need to see, editing the code to add a print statement, rerunning, and then remembering to remove it afterward — and every new question ("what about this other variable?") means repeating the whole cycle. The visual debugger inspects *anything* at the paused moment — no code changes, no reruns, and stepping through lets you follow the program's actual logic interactively rather than guessing where to place the next print statement.

⚠ Breakpoints Only Work Through the Debugger

Running a file normally (via the integrated terminal's `python script.py`, for instance, rather than pressing `F5`) completely ignores every breakpoint — they're a debugger-only feature, not something baked into the file itself. If breakpoints seem to be "not working," the first thing to check is whether the program was actually started via **Run & Debug** at all. For compiled or transpiled languages (TypeScript compiling to JavaScript, for example), breakpoints set in your original source also require *source maps* to correctly map back from the running compiled code — worth knowing this exists as a common source of "the breakpoint just doesn't hit" confusion in those setups.

💻 Coding Challenges

Exercise 1: Set Up launch.json and Hit a Breakpoint

In any small script, create a basic `launch.json` for your language, set a breakpoint on any line, and press `F5`. Confirm execution actually pauses there.

→ Solution

Exercise 2: Fix This Chapter's average() Bug

Using this chapter's worked example, set a breakpoint and use the debugger to confirm the `- 1` is the bug, then fix the function so it returns the correct average.

→ Solution

Exercise 3: A Conditional Breakpoint in a Loop

Write a loop that runs at least 10 iterations. Set a conditional breakpoint inside it that only pauses on a specific iteration (e.g. when the loop variable equals 7), and confirm it doesn't pause on any earlier iteration.

→ Solution

🎯 What's Next

Next chapter: **Snippets & Keybindings** — writing custom snippets, remapping keys, and syncing settings across machines.

⌨️ Snippets & Keybindings

Chapter 5 covered the settings a project shares with a whole team. This chapter covers the opposite end — personalizing VS Code around your own habits: custom code snippets, remapped keys, and keeping all of it in sync across every machine you use.

Built-In Snippets

Most language extensions (Chapter 4) already ship snippets — in a JavaScript file, typing `for` and pressing `Tab` expands a full `for`-loop skeleton, with `Tab` jumping between its placeholders (the loop variable, the condition, the body) one at a time.

✍️ Writing a Custom Snippet

The Command Palette's (Chapter 2) **Preferences: Configure User Snippets** command lets you pick a language (or a global scope covering every file type) and opens a JSON file defining your own:

```
// python.json (User Snippets for Python)
{
 "Function with docstring": {
  "prefix": "defdoc",
  "body": [
   "def ${1:function_name}(${2:args}):",
   "    \"\"\"${3:Describe what this function does.}\"\"\"",
   "    $0"
  ],
  "description": "Function skeleton with a docstring"
 }
}
```

`prefix` is what you type to trigger it; `body` is an array of lines (each array element becomes one line, avoiding awkward embedded newline characters). `${1:function_name}` is a **tab stop** with default placeholder text — pressing `Tab` after inserting the snippet jumps to it, pre-selected and ready to type over; `$2`, `$3` are further stops in order; `$0` marks where the cursor lands *last*, after every other stop has been filled in.

🔧 A Worked Example

Typing `defdoc` then `Tab` in a Python file expands to:

```
def function_name(args):
    """Describe what this function does."""
```

...with `function_name` already selected, ready to type a real name. Pressing `Tab` again jumps to `args`, then to the docstring text, then finally to the blank body line (`$0`) — a full function skeleton assembled in one trigger and a few `Tab` presses, instead of typing every character by hand.

⌨️ Keybindings — Remapping Keys

`Ctrl+K Ctrl+S` (press `Ctrl+K`, release, then `Ctrl+S`) opens the **Keyboard Shortcuts** UI — searchable by command name, using the exact same command names the Command Palette (Chapter 2) already uses. Hovering a command reveals a pencil icon to add or change its shortcut. For raw editing, **Preferences: Open Keyboard Shortcuts (JSON)** opens `keybindings.json` directly:

```
// keybindings.json
[
 {
  "key": "ctrl+alt+l",
  "command": "editor.action.formatDocument",
  "when": "editorTextFocus"
 }
]
```

`when` is a **context clause** — it restricts a binding to specific situations (here, only while actually focused in the editor), preventing it from firing unexpectedly in, say, the terminal or the Settings UI.

⚠ Check for Conflicts Before Remapping a Key

A key combination already bound to something else doesn't automatically get freed up just because you add a new binding for it — depending on `when` clauses, you can end up with two commands competing for the same key, one silently winning. The Keyboard Shortcuts UI flags this directly with a visible "conflict" indicator next to any shortcut that's bound more than once in overlapping contexts — always check for it before assuming a new binding is safe.

☁ Syncing Settings Across Machines

Settings Sync (built in, no extension needed) keeps settings, keybindings, snippets, installed extensions, and general UI state synchronized across every machine you use it on. Turn it on via the accounts icon in the Activity Bar's bottom-left corner, or the Command Palette's **Settings Sync: Turn On** — sign in with a Microsoft or GitHub account, and everything from this chapter follows you to a new machine automatically.

Personal vs. Team-Shared Configuration

Chapter 5 drew a line between User settings (personal) and Workspace settings (shared, committed to the repo). Snippets and keybindings sit firmly on the personal side of that line — they're about individual muscle memory and workflow, not a project-wide standard, which is why Settings Sync (this chapter) targets your *account*, not a *repository*. A team's shared `.vscode/extensions.json` (Chapter 5) says "everyone on this project should have ESLint"; your own `keybindings.json` says nothing about the project at all — it's entirely about you.

Snippet body & tab stops

`$1`, `$2`… in order, `$0` last — jump between placeholders with `Tab`.

Keyboard Shortcuts UI

`Ctrl+K Ctrl+S` — search by command name, pencil icon to rebind.

when clauses

Restrict a keybinding to a specific context, avoiding accidental overlap.

Settings Sync

Account-based, cross-machine sync for settings, keybindings, snippets, and extensions.

💻 Coding Challenges

Exercise 1: Write Your Own Snippet

Using **Preferences: Configure User Snippets**, write a snippet for any language you use, with at least 2 tab stops. Trigger it in a real file to confirm it expands correctly.

→ Solution

Exercise 2: Rebind a Command

Open the Keyboard Shortcuts UI and rebind any command to a key combination of your choice, checking for a conflict indicator first. If one appears, pick a different combination.

→ Solution

Exercise 3: Turn On Settings Sync

Turn on Settings Sync via the accounts icon or the Command Palette, and confirm in its menu which categories (settings, keybindings, snippets, extensions) are currently enabled for syncing.

→ Solution

🎯 What's Next

Final chapter: **Remote Development** — Remote-SSH, Dev Containers, and WSL integration.

🌐 Remote Development

The final chapter of VS Code Essentials — and a genuine capstone of sorts, drawing directly on the site's own SSH (`ssh1`) and Docker (`docker1`/`docker2`) courses. Remote Development lets VS Code's actual editing environment run somewhere other than your local machine — a remote server, a container, or a Linux subsystem — while everything still feels like a normal local window.

🏗️ The Core Idea — A Thin Client, A Remote Server

A lightweight VS Code client runs locally — the window, the UI, everything from Chapters 1-9. But a small **VS Code Server** component runs on the *remote target* — the SSH host, the container, the WSL instance. Extensions that need to interact with your code (a language server, a debugger, Chapter 4/8's tooling) run *on the remote side* too, so they see the real files, the real installed dependencies, and the real environment — not a local guess at what's over there.

🔑 Remote-SSH

The **Remote - SSH** extension connects to any machine you can already reach via plain SSH — reusing the exact same hosts, keys, and config from the site's own `ssh1` course (`~/.ssh/config` entries appear automatically in the Remote Explorer view). Connecting opens a new window where the Explorer, terminal (Chapter 6), and debugger (Chapter 8) all operate directly on the remote machine — editing a file there feels identical to editing one locally, but the file itself never left the remote server.

📦 Dev Containers

The **Dev Containers** extension goes a step further: instead of connecting to a persistent server, it defines a whole disposable development environment as code, in `.devcontainer/devcontainer.json`:

```
// .devcontainer/devcontainer.json
{
 "name": "Python Dev Container",
 "image": "mcr.microsoft.com/devcontainers/python:3.12",
 "customizations": {
  "vscode": {
   "extensions": ["ms-python.python"]
  }
 }
}
```

The Command Palette's (Chapter 2) **Dev Containers: Reopen in Container** builds (or pulls, per the site's own `docker1` course) the specified image and reopens your project running entirely inside it — every teammate who opens the same repo gets the exact same base image, tools, and even auto-installed extensions, regardless of what's actually installed on their own host machine.

WSL Integration

On Windows specifically, **Remote - WSL** connects to the Windows Subsystem for Linux — a real Linux environment running alongside Windows, no network or virtual machine involved. Opening a folder that lives inside the WSL filesystem gives you a genuine Linux terminal, Linux-native tooling, and Linux file permissions, all through the same familiar Windows-hosted VS Code window — useful for Linux-targeted development without dual-booting or running a full VM.

Choosing Between the Three

Target	What it actually is	Best for
Remote-SSH	A real, persistent remote machine (a cloud server, a beefier desktop)	Working on hardware more powerful than your laptop, or code that must live on a specific server
Dev Containers	A container, defined entirely by config, disposable and rebuildable	Guaranteed-identical environments across a whole team — the most reproducible option
WSL	A Linux environment inside Windows itself — no network involved	Linux-native tooling on a Windows machine, with none of Remote-SSH's network dependency

🎯 Why This Matters

Remote Development directly attacks the "works on my machine" problem — the same problem the site's `docker2` capstone (productionizing a multi-container app) addresses from the deployment side. Here, the same reproducibility guarantee applies one step earlier, to development itself: a Dev Container defined once in a repo means no teammate ever debugs a bug that only exists because of their own machine's particular installed versions.

⚠ Remote Access Is Still Real Access — With Real Permissions

Connecting to a new SSH host for the first time triggers a host-key fingerprint prompt — the same trust-on-first-use mechanism from the site's own `ssh1` course. Never blindly accept a fingerprint that unexpectedly *changes* for a host you've connected to before; that's a classic sign of a potential man-in-the-middle attack, not routine behavior. And remember Chapter 4's extension warning applies doubly here: a remote session runs a full VS Code Server with real permissions on that remote machine — the same evaluation discipline (trusted extensions, verified hosts) matters even more once a remote system, not just your local one, is on the line.

VS Code Server

Runs on the remote target; extensions and tooling execute there, not locally.

Remote-SSH

Connects to any real machine reachable via SSH, reusing `ssh1`'s config and keys.

Dev Containers

`devcontainer.json` defines a reproducible, disposable environment as code.

WSL

A real Linux environment inside Windows — no network, no VM.

💻 Coding Challenges

Exercise 1: Explore the Remote Explorer

Install the "Remote Development" extension pack, open the Remote Explorer view, and check its SSH Targets tab. If you have an existing `~/.ssh/config` from the site's `ssh1` course, note whether its hosts already appear here automatically.

→ Solution

Exercise 2: Write a devcontainer.json

For any small project, write a `.devcontainer/devcontainer.json` specifying a base image appropriate to that project's language, plus at least one recommended extension. If Docker is installed, use "Dev Containers: Reopen in Container" to actually try it.

→ Solution

Exercise 3: Match the Scenario to the Right Tool

For each of these three scenarios, decide which remote option (Remote-SSH, Dev Containers, or WSL) fits best, and explain why: (a) a five-person team needs identical environments regardless of each member's own OS, (b) a data scientist needs a specific GPU physically installed in a lab server, (c) a Windows user wants to run Linux-only build tools without a VM.

→ Solution

🎓 Course Complete!

That's all 10 chapters of **VS Code Essentials** — the interface, keyboard-first navigation, efficient editing, extensions, workspaces and settings, the terminal and tasks, git integration, debugging, snippets and keybindings, and remote development. Everything from here builds naturally on the language- and tool-specific courses already on the site — the editor is now yours to actually work in.