

FREE SAMPLE · CHAPTERS 1–5

SHELL SCRIPTING MANUAL

Bash, Distilled

— Not Stirred —

The Practical, Production-Grade Bash Manual

The first five chapters of Bash, Distilled — Not Stirred, complete and unabridged: what a terminal actually is, finding your way around the filesystem, writing your first script, the shebang line explained, and how Bash actually differs from sh and Zsh. The full book continues from Chapter 6 through arrays, defensive scripting, real-world automation, testing, and a full quick-reference cheat sheet.

- ✓ Zero to production, step by step
- 🛡 Security for privileged scripts
- 📦 Bash in containers & CI/CD
- ⚙ Tested, linted & portable
- 🔄 Real backups, cron & automation
- 📖 Full quick-reference cheat sheet

5

CHAPTERS HERE

31

CHAPTERS TOTAL

135

PAGES TOTAL

Linux (Ubuntu, Debian & beyond)

Windows 11 (via WSL)

macOS 26 (Tahoe)

*SplitZrate.com*Free sample · Chapters 1–5 of 31 · get the full book at SplitZrate.com

PRACTICAL RELIABLE NO FLUFF

Contents

Part I — Getting Oriented

01 What a Terminal Actually Is (and How to Open One) IN THIS SAMPLE	02 Finding Your Way Around: Files, Directories & Basic Commands IN THIS SAMPLE
03 From Typing Commands to Writing Scripts IN THIS SAMPLE	

Part II — Bash Foundations

04 The Shebang Line, and Why It's Not Optional IN THIS SAMPLE	05 Bash vs. sh vs. Zsh: What Actually Differs IN THIS SAMPLE
06 Running a Script: chmod, PATH, and the Three Ways to Invoke It	07 Variables & the Quoting Rules That Actually Matter
08 Conditions, Tests & Case	09 Loops & Safe Iteration
10 Functions, Scope & Return Values	

Part III — Data, Text & Expansion

11 Arrays & Associative Arrays	12 Parameter Expansions
13 Globbing & Pattern Matching	14 Arithmetic & Numbers
15 Regular Expressions in Bash	16 Command & Process Substitution

Part IV — Writing Scripts That Don't Break

17
Defensive Scripting: set -euo pipefail and
Its Limits

19
Debugging Bash Scripts

21
Logging & Output Done Right

18
Error Handling, Traps & Cleanup

20
Argument Parsing & User Input

Part V — Real-World Automation

22
Automating Backups & File Tasks: A Full
Walkthrough

24
Talking to APIs: curl, jq & JSON in Bash

23
Cron, systemd Timers & Long-Running
Scripts

25
Bash in Containers & CI/CD

Part VI — Professional Practice

26
Testing Bash Scripts with Bats

28
Portability: Bash vs. POSIX sh

30
When Bash Isn't the Answer (and What Is)

27
Linting, Style & ShellCheck

29
Security for Privileged Scripts

31
Quick-Reference Cheat Sheet

This sample contains Chapters 1–5, complete and unabridged. Chapters 6–31 (grayed out above) are in the full book — get it at SplitZrate.com.

1 What a Terminal Actually Is (and How to Open One)

If you've never typed a command in your life, you're in exactly the right place. This chapter assumes nothing going in — no terminal experience, no familiarity with Bash. If "the command line" doesn't mean anything to you yet, that's exactly the point. By the end, you'll have one open, and you'll have typed into it and survived.

1.1 Graphical Interface vs. Command Line: Two Ways to Talk to a Computer

Almost everything you've done on a computer so far has probably gone through a **graphical interface** — icons, windows, buttons, a mouse pointer. You click a folder to open it, drag a file to move it, and the computer shows you a picture of what's happening.

A **command line** is the other way to talk to the same computer: instead of clicking, you type a short line of text describing what you want, press Enter, and the computer does it — then tells you what happened, in text. There's no picture of a folder to drag; there's just a word for "folder" (technically a **directory**, a term Chapter 2 explains properly) and a command that moves it.

Neither approach is "the real one" and neither is obsolete — they're two interfaces to the same underlying system, and most people who work in tech comfortably use both, switching between them depending on the task. What the command line offers that a mouse mostly can't is **repeatability**: a line of text you typed once can be saved, reused, and run a thousand times identically. That reusability is the entire reason this book exists — it's what turns "things I do by hand" into "things a script does for me," which Chapter 3 gets into directly.

1.2 What a Shell Prompt Is (and How to Read One)

The program that reads what you type, runs it, and shows you the result is called a **shell**. When a shell is ready for your next command, it shows a **prompt** — a short line, usually ending in `$` or `%`, that means "I'm listening." A typical one looks like this:

```
michael@laptop:~$
```

Reading that left to right: `michael` is the username you're logged in as, `Laptop` is the computer's name, `~` is shorthand for "your home directory" (Chapter 2 explains exactly what that means), and `$` marks the actual spot where your typing begins. Everything before the `$` is just information the prompt is showing you — you never type any of it yourself.

NOTE

Prompts look slightly different on every system, and that's fine — some show the current time, some show a folder icon, some are a single bare `$` with nothing else. Whatever your prompt looks like, the rule is the same: it appears when the shell is ready, and it disappears while a command is still running.

1.3 Opening a Terminal: Windows, macOS & Linux

The window that runs a shell for you is called a **terminal** (or "terminal emulator," if you want the precise name — a nod to the physical terminals this all descends from). Every major operating system ships one:

System	How to open it	Default shell
Linux	Search for "Terminal" in your applications menu, or press <code>Ctrl+Alt+T</code> on many distros	Bash (usually)
macOS	Press <code>Cmd+Space</code> , type "Terminal", press Enter	Zsh (see Chapter 5)
Windows	Search for "Windows Terminal" or "WSL" in the Start menu	Depends on setup — Chapter 5 covers this

On Windows specifically, a real bash environment usually means installing WSL (Windows Subsystem for Linux) first — a genuine Linux system running alongside Windows. That's a one-time setup step this book won't walk through in detail, since Microsoft's own installation docs stay current more reliably than a printed page can; search "install WSL" and you'll find an up-to-date, official walkthrough.

Whichever system you're on, open a terminal now, before reading any further. The rest of this chapter is meant to be typed along with, not just read.

CROSS-PLATFORM NOTE

Every path shown here is written the Linux/macOS way, with forward slashes — `/home/michael/notes`, not `C:\Users\michael\notes`. That's correct whether you're on native Linux, macOS, or WSL on Windows: inside a bash shell, paths always use forward slashes, even on Windows. You'll only see backslash paths if you step outside bash into PowerShell or `cmd.exe` — something none of this asks you to do.

1.4 Your First Commands: `pwd`, `whoami`, `date`, `echo`

With a terminal open and a prompt waiting, type each of these, pressing Enter after each one:

```
# where am I, right now?
$ pwd
/home/michael

# who am I logged in as?
$ whoami
michael

# what time does the computer think it is?
$ date
Wed Aug 19 14:32:07 CEST 2026

# say something back
$ echo hello
hello
```

Each command follows the exact same pattern: you type a word (the command's name), you press Enter, and the shell prints back a line or two of text before showing you a fresh prompt. `pwd` ("print working directory") tells you which directory you're currently in; `whoami` prints your username; `date` prints the current date and time; and `echo` simply repeats back whatever you typed after it. None of these four commands change anything on your computer — they only report information back to you, which makes them a safe, honest way to get comfortable with the rhythm of typing a command and reading its response.

WHAT THIS DOES

Each line above is one command, typed and run on its own — not a script, nothing saved to a file. Your screen will show slightly different text (your own username instead of `michael`, today's real date and time instead of this example), but the shape is identical: type it, press Enter, read the line it prints back. There's nothing to set up first and nothing that can go wrong here — if you have a terminal open already, feel free to type all four right now and watch your own computer answer back.

1.5 Mistakes Are Free: Why You Can't Break Anything Just by Typing

The single biggest thing stopping people from experimenting at a command line is a fear that got planted somewhere — that one wrong word could break the entire computer. It's worth addressing directly, before it silently limits how much you're willing to try.

Simply **typing** a word and pressing Enter is safe. If you misspell a command, the shell doesn't guess and run something else — it tells you plainly that it doesn't recognize what you typed, and nothing happens:

```
$ pwdd
pwdd: command not found
```

That response — `command not found` — is one of the most common things you'll see as a beginner, and it means exactly what it says: nothing ran, nothing changed, no harm done. The shell will never run a command you didn't actually type, and it will never guess at what you "probably meant."

TIP

The genuine risk in a terminal isn't typos — it's commands that **delete or overwrite** something, run correctly, and do exactly what you told them to. Every one of those gets a clear warning box the moment it shows up — the first is `rm`, two chapters from now. Until then, everything in this chapter is exploration-safe.

CHAPTER 1 — QUICK RECAP

- A terminal is a window running a **shell** — a program that reads a typed command, runs it, and prints the result.
- The **prompt** (often ending in `$`) means the shell is ready for your next command.
- Open a terminal: Linux (applications menu or `Ctrl+Alt+T`), macOS (`Cmd+Space` → "Terminal"), Windows (install WSL first, then Windows Terminal).
- Four safe first commands: `pwd`, `whoami`, `date`, `echo` — none of them change anything.
- Typing something wrong just gets you `command not found` — mistakes from typing alone are free. The real risk is commands that intentionally delete or overwrite something, flagged explicitly with a warning box from Chapter 2 onward.

2 Finding Your Way Around: Files, Directories & Basic Commands

You can open a terminal and type a few safe commands. This chapter gives you a mental map of what's actually out there to look at — how a computer organizes every file it has — and the handful of commands you'll use constantly to move around, look around, and reorganize what you find.

2.1 The Filesystem as a Tree: Directories, Not "Folders"

Every file on your computer — every document, photo, program, and setting — lives somewhere inside one enormous structure, and that structure is a tree. At the very top sits a single starting point; everything else branches downward from it. A graphical interface shows you this as "folders" nested inside other folders, complete with little folder icons. At the command line the correct term is **directory** — the same idea, a different name, and the name every command and every error message will actually use from here on.

On Linux and macOS, that single starting point is called the **root directory**, written as just a forward slash: `/`. Every file and directory on the whole system, no matter how deeply nested, can be traced back up to that one root. Windows organizes things differently — separate drive letters (`C:\` , `D:\`) instead of one shared root — which matters the moment you're running bash inside WSL, since WSL has to reconcile the two systems for you.

■ CROSS-PLATFORM NOTE

Inside a WSL terminal on Windows, you're working in a real Linux filesystem with its own single root (`/`) and its own home directory (`/home/yourname`) — separate from Windows entirely. Your Windows drives aren't gone; WSL mounts them under `/mnt` , so `C:\Users\michael` becomes `/mnt/c/Users/michael` from inside bash. Almost every example here lives under `/home/yourname` ; `/mnt/c` only comes up when a script specifically needs to reach a file on the Windows side.

2.2 Absolute vs. Relative Paths

A **path** is just an address describing where a file or directory sits inside the tree. Paths come in two flavors. An **absolute path** always starts from the root and spells out the full route — it describes the same location no matter where you currently happen to be. A **relative path** is written starting from wherever you currently are, which is shorter to type but only means what you think it means if you know your current location.

```
# absolute — works from anywhere
/home/michael/notes/todo.txt

# relative — only correct if you're already inside /home/michael
notes/todo.txt
```

Two shorthand relative paths come up constantly: a single dot `.` means "the directory I'm in right now," and two dots `..` means "one directory up from here." Both are used far more as ingredients in other commands than on their own — the next section puts them to work.

2.3 Moving Around: `cd`, `pwd`, and `~`

`cd` ("change directory") moves you somewhere else in the tree, and accepts either kind of path from the last section. `pwd` — already familiar from Chapter 1 — always tells you exactly where that move landed you, which makes the two a natural pair whenever you're unsure.

```
pwd
/home/michael

cd Documents
pwd
/home/michael/Documents

cd ..
pwd
/home/michael
```

One more shorthand, and it's the most useful of the three: `~` (tilde) always means **your home directory** in any command, from anywhere — not just at the start of the prompt where Chapter 1 first pointed it out. Typing `cd` alone, with no path at all, does the same thing as `cd ~`: it takes you home. And `cd -` (a dash) jumps back to whichever directory you were in immediately before your last `cd` — a fast way to bounce between two locations without retyping either path.

2.4 Looking Around: `ls` and Its Most Useful Flags

`ls` lists whatever is inside the directory you're currently in. On its own it's already useful, but a handful of flags — single letters added after a dash — make it tell you considerably more.

Flag	What it adds
<code>-l</code>	Long format: permissions, owner, size, and last-modified date for each entry, one per line
<code>-a</code>	Shows hidden entries too — anything starting with a dot, like <code>.bashrc</code>
<code>-h</code>	Human-readable sizes (<code>4.2K</code> , <code>1.1M</code>) instead of a raw byte count — only means something combined with <code>-l</code>

Flags combine, so `ls -la` and `ls -lah` are both extremely common in practice — full detail with hidden files included, in a size format a human can actually read at a glance. A file or directory starting with a dot isn't secret or broken; hiding config files from a plain `ls` by convention is just how Linux and macOS both keep everyday listings uncluttered.

2.5 Creating, Copying, Moving & Removing: `mkdir`, `cp`, `mv`, `rm`

Four commands actually change what's on disk, rather than just looking at it. Three of them are completely safe to experiment with. The fourth needs real respect.

```
# make a new directory
mkdir drafts

# copy a file – original stays where it was
cp notes.txt drafts/notes.txt

# move a file – or rename it, which is the same operation
mv drafts/notes.txt drafts/notes-old.txt

# remove a file – permanently
rm drafts/notes-old.txt
```

`mv` covers two jobs that feel different but are technically identical: moving a file to a new location, and renaming a file in place. Under the hood, renaming *is* moving — you're just moving something to a new name in the same directory instead of to a different directory.

WHAT THIS DOES

Notice that none of the four commands above printed anything back. That's normal, not a problem — `mkdir`, `cp`, `mv`, and `rm` all follow the same unwritten rule as most command-line tools: silence means success. If something had gone wrong — the directory already existed, the file wasn't found — you'd see an error message instead. No news really is good news here. If you want to see the proof for yourself, run `ls drafts` after the sequence above and you'll find `notes-old.txt` sitting there, exactly where the commands left it.

⚠ WARNING

`rm` deletes immediately and permanently. There is no Recycle Bin, no Trash, no "are you sure?" prompt to catch you — unlike deleting a file in a graphical file manager, this cannot be undone by opening a bin and dragging the file back. Typing `rm notes.txt` when you meant `rm notes.txt.bak` removes the wrong file with zero recovery. This is the first genuinely dangerous command covered so far — a mistake with real, permanent consequences, not just an error message. Later chapters cover safer patterns (backups, confirmation flags, dry runs); until then, the simplest protection is the oldest one: read the command back, in full, before you press Enter.

2.6 Tab Completion: Let the Shell Finish Your Sentences

Every command and path in this chapter can be typed faster, and more safely, using **tab completion**. Start typing a command or a file name, press `Tab`, and the shell fills in as much of the rest as it can figure out unambiguously. If more than one match is possible, pressing `Tab` a second time lists every option, so you can type one more distinguishing letter and try again.

```
cd Doc[press Tab]
cd Documents/ ← filled in automatically
```

Beyond the convenience, tab completion is a genuine safety net: a path that doesn't exist simply won't complete, which catches a typo before you ever press Enter on it. It's also entirely a feature of the shell itself, not the operating system — it behaves identically whether you're on native Linux, macOS, or WSL on Windows.

CHAPTER 2 — QUICK RECAP

- Every file lives inside one branching **directory** tree; Linux and macOS share a single root (`/`), Windows uses drive letters, and WSL bridges the two by mounting Windows drives under `/mnt`.
- **Absolute paths** start from the root and work from anywhere; **relative paths** start from your current directory.
- `cd` moves you, `pwd` confirms where you landed, `~` always means your home directory, `cd -` jumps back to your previous location.
- `ls -la` (or `-lah`) is the combination you'll use constantly: full detail, hidden files included, readable sizes.
- `mkdir`, `cp`, and `mv` are safe to experiment with. `rm` is not reversible — read the command back before pressing Enter.
- **Tab completion** speeds up typing and catches typos in paths before they run.

3 From Typing Commands to Writing Scripts

Everything so far has been one command at a time, typed by hand into a waiting prompt. This chapter makes the jump from that to a **script** — a file holding a sequence of commands the shell can run together, in order, without you retyping a word. It's a small jump technically, but it's the one that turns the command line from a way to look things up into a tool you build with, which is what the rest of this book is actually about.

3.1 What "Bash" Actually Means: Shell, Script & Language

"Bash" gets used loosely to mean three related things at once, and it's worth separating them now. It's the **shell** — the program reading what you type at the prompt and running it, exactly as Chapter 1 described. It's also a full **programming language**, with variables, conditions, and loops, covered in depth across Parts II through IV. And a **bash script** is simply a plain text file containing a sequence of bash commands, saved so the shell can run all of them together instead of you typing each one by hand.

That last point matters more than it looks: every command you've already typed in Chapters 1 and 2 — `pwd`, `ls`, `cd`, `mkdir` — is already valid bash. A script isn't a different, harder language to learn on top of what you know. It's the same words you've already typed, written down in a file instead of into a prompt.

3.2 Everything You Just Did by Hand, a Script Can Do for You

Picture a small routine: check where you are, list what's in a folder, create a new subdirectory for today's work, move into it. Four commands, typed by hand, every single time you start. That's tedious on the tenth day, and — per Chapter 1's warning about typos — exactly the kind of repetition where a mistyped command eventually slips through.

A script runs that same sequence identically every time, in the same order, without retyping any of it. This is the **repeatability** Chapter 1 raised back in section 1.1 as the whole reason the command line offers something a mouse mostly can't: a sequence you write once can be saved, reused, and run a thousand times without drifting. Section 3.3 turns that idea into an actual file.

3.3 Your First Script: Three Lines That Actually Do Something

Creating a script starts with a text editor. Any editor works, but these examples use `nano`, since it opens directly inside the terminal you already have open, with no separate program to launch:

```
nano hello.sh
```

Type the following three lines into it exactly, then save and exit (nano shows the keys for both at the bottom of the screen):

```
#!/bin/bash
# my first script
echo "Hello, $(whoami). Today is $(date +%A)."
```

The very first line, starting with `#!/`, is called a **shebang** — it tells the system which program should actually run this file. Chapter 4 explains exactly why that line matters and what happens if it's wrong; for now, always include it exactly as shown, as the first line and nothing before it.

Run the script the simplest way — Chapter 6 covers two more ways and when each applies:

```
bash hello.sh
Hello, michael. Today is Wednesday.
```

The `$(...)` parts run `whoami` and `date` exactly as you typed them yourself in Chapter 1, and drop each command's own output directly into the middle of the text — which is how a script ends up saying something different every time it runs, using data it looked up itself instead of text you hardcoded.

WHAT THIS DOES

This is the whole thing: three lines, saved as a file, run once. The result is a single line of greeting text printed to your screen — nothing on your computer is changed, saved, or deleted, and nothing runs in the background afterward. It's exactly as safe as typing `echo` directly at the prompt, the way you did in Chapter 1; the only difference is that this time the commands live in a file instead of being typed fresh each time. If you have a terminal open, this is a good moment to actually create `hello.sh` and run it yourself rather than just reading along — and if you'd rather keep reading first and try it later, that works too. Either way, the first time you see your own username and today's real date print back at you, you've written and run a working script.

3.4 Comments, and Documenting a Script for Future You

Any line starting with `#` is a **comment** — bash ignores it completely when running the script. The `# my first script` line above is exactly this: it does nothing when the script runs, and exists purely for whoever reads the file afterward — very often you, months later, having forgotten why a line does what it does.

A comment earns its place when it explains *why* a line exists, not *what* it does — the "what" is usually clear enough from the command itself. As scripts grow longer in later chapters, this habit is what keeps them readable instead of becoming a wall of commands only their original author could ever untangle.

3.5 Where This Book Goes From Here

Part I ends here. In three chapters, that's a terminal opened, a filesystem navigated with real understanding of paths, and a working script saved to disk and run successfully. Part II starts immediately from this point: Chapter 4 explains that shebang line properly, and Chapter 5 gets a real, working bash running consistently on Windows and macOS — not just Linux, where it's already the default. By Chapter 10, the same three-line script above has grown into scripts with variables, conditions, loops, and functions. None of it requires anything beyond what this chapter already used; it's the same handful of ideas, applied with more precision.

CHAPTER 3 — QUICK RECAP

- "Bash" means three things: the **shell** you type into, a full **programming language**, and a **script** — commands saved to a file instead of typed by hand.
- Every command used in Chapters 1–2 is already valid bash — a script is those same commands, saved.
- A script's first line is a **shebang** (`#!/bin/bash`) — covered properly in Chapter 4.
- Run a script with `bash scriptname.sh` — two more ways follow in Chapter 6.
- Lines starting with `#` are **comments**: ignored by bash, written for whoever reads the script later.
- Part I is complete. Part II (Chapters 4–10) starts immediately, building on exactly what's been used so far.

4 The Shebang Line, and Why It's Not Optional

Every script so far has started with the same cryptic first line — `#!/bin/bash` — typed without much explanation. Here's what the line actually does, why it has to be exactly right, and what happens when it isn't.

4.1 The First Line of Every Script

That first line is called a **shebang** (from "hash-bang" — `#` plus `!`), and it's not a bash feature at all — it's a Linux and macOS kernel feature. When you run a file directly (a form Chapter 6 covers properly), the kernel opens it, looks at the very first two characters, and checks whether they're `#!`. If they are, everything after it on that line is treated as the path to a program, and the kernel hands your file to that program to actually run — instead of trying to execute your text as a compiled binary, which would just fail.

```
#!/bin/bash
```

Read literally: "use the program at `/bin/bash` to run this file." That's the entire mechanism. It has to be the first line, and `#!` has to be the first two characters on it — no blank line above it, no leading space before the `#`. The kernel doesn't search for a shebang further down the file; it only ever looks at the start.

4.2 Hardcoded Path vs. env Lookup

`#!/bin/bash` hardcodes an exact location: it only works if bash genuinely lives at `/bin/bash` on the machine running it. On most Linux systems it reliably does. There's a second, more defensive form you'll see just as often:

```
#!/usr/bin/env bash
```

Instead of hardcoding a location, this runs the `env` program, which searches `PATH` — the same list of folders the shell itself searches when you type a command name — and uses whichever `bash` it finds first. That indirection matters more than it looks: it's the difference between "the bash at this exact spot" and "whatever bash you'd get if you typed `bash` right now."

■ CROSS-PLATFORM NOTE

This isn't a theoretical distinction on macOS. Apple ships a genuinely ancient bash 3.2 at `/bin/bash` for licensing reasons and hasn't updated it in over a decade — several examples later on simply won't run on it. A modern bash installed via Homebrew lands somewhere else entirely (typically `/opt/homebrew/bin/bash`), so `#!/bin/bash` on macOS can silently mean "the old one" even after you've installed a new one. `#!/usr/bin/env bash` finds whichever bash comes first on `PATH` instead — which is exactly why it's the default from here on. Chapter 5 walks through getting a modern bash installed and correctly first on `PATH`, on both macOS and Windows (WSL).

`#!/usr/bin/env bash` is the default going forward for exactly that reason. Chapters 1 through 3 used the hardcoded form because it's one fewer new idea while you were still getting used to a terminal at all — both are correct bash, and you'll see plenty of real-world scripts using either.

4.3 When the Shebang Matters (and When It's Ignored)

The shebang line only does anything when the *kernel* is the one reading it — which only happens when a script is run directly. If you instead run a script by naming `bash` explicitly, as every example so far has done, bash itself opens the file and reads it top to bottom as bash commands. It never asks the kernel to interpret the first line at all — and since a line starting with `#` is just a comment (Chapter 3.4), bash silently treats the shebang as exactly that: a comment it skips right over.

```
# hello.sh, with a typo in the shebang
#!/bin/bsah
echo "Hello, $(whoami)."
```



```
$ bash hello.sh
Hello, michael. ← still works – bash never looked at that line as anything but a
comment
```

Run the exact same broken file *directly*, though, and the kernel is now the one reading that first line — and it fails, because there's no program at `/bin/bsah`:

```
$ ./hello.sh
bash: ./hello.sh: /bin/bsah: bad interpreter: No such file or directory
```

TIP

`bad interpreter` is always about the shebang line specifically — not the rest of your script, however long or broken it might separately be. If you ever see that exact message, the fix is almost always a typo or a wrong path on line one, and nowhere else.

4.4 Making a Script Executable

Running a script directly (`./hello.sh`) needs one more thing besides a correct shebang: permission to execute it as a program in the first place. By default, a newly created file doesn't have that permission, even if you own it and can freely edit its contents:

```
$ ./hello.sh
bash: ./hello.sh: Permission denied
```

`chmod +x` ("change mode, add execute") grants it:

```
$ chmod +x hello.sh
$ ./hello.sh
Hello, michael.
```

NOTE

This permission is only checked when running a file directly. Running the same unmodified file with `bash hello.sh` works with or without `chmod +x`, since you're telling bash to read the file's contents, not asking the kernel to execute it as a standalone program. Chapter 6 covers when each way of running a script actually matters.

CHAPTER 4 — QUICK RECAP

- The **shebang** (`#!/path`) is a kernel feature, not a bash one — it only matters when a script is run directly.
- `#!/bin/bash` hardcodes a path; `#!/usr/bin/env bash` searches `PATH` instead — the `env` form is the default from here on, especially relevant on macOS.
- Running a script via `bash scriptname.sh` ignores the shebang entirely — bash treats it as an ordinary comment.
- A broken shebang only fails when running a script *directly*, producing `bad interpreter`.
- `chmod +x` grants execute permission, required only for running a script directly — not for `bash scriptname.sh`.

5 Bash vs. sh vs. Zsh: What Actually Differs

"Just open a terminal and type bash commands" silently glosses over something Chapters 1 through 4 sidestepped: the shell you land in by default isn't always bash, and the bash you do have isn't always the same bash. This chapter clears both up — and walks through actually getting a modern, working bash on every supported platform, start to finish.

5.1 /bin/sh Is Not Bash on Debian and Ubuntu

`/bin/sh` looks like it should just be another name for bash — plenty of older scripts use `#!/bin/sh` as their shebang. On Debian and Ubuntu, though, `/bin/sh` is a symlink to **dash** (the Debian Almquist Shell), a much smaller shell that implements only the POSIX standard — not bash's extensions on top of it. This is exactly the scenario Chapter 4 warned about: the shebang decides which interpreter actually runs your file, and getting it wrong doesn't always fail loudly.

```
# check.sh
#!/bin/sh
if [[ "$1" == "yes" ]]; then
    echo "confirmed"
fi
```

```
$ ./check.sh yes
./check.sh: 2: [: not found
```

`[[ ]]` is a bash extension — dash has never heard of it, and the error appears deep inside the script instead of on line one, which makes it easy to misdiagnose as a logic bug rather than a shebang problem. The fix, consistent with Chapter 4: if you want bash, ask for bash explicitly.

TIP

Use `#!/usr/bin/env bash` whenever you're deliberately writing bash — which is every script here. Reach for `#!/bin/sh` only when you specifically want POSIX-portable behavior and are avoiding bash-only features on purpose; Chapter 28 covers that trade-off properly, once you've seen enough bash-only features to know what you'd actually be giving up.

5.2 macOS's Default Shell Is Zsh, Not Bash

Since macOS Catalina (2019), the default interactive shell — what opens when you launch Terminal.app, as Chapter 1 described — is **zsh**, not bash. Apple made the switch because newer bash versions are licensed under the GPLv3, a license Apple avoids shipping; zsh has no such restriction. This matters less than it sounds like it should: every script here is invoked through an explicit `#!/usr/bin/env bash` shebang or a direct `bash` command, so your default interactive shell never enters the picture. You do not need to change it, and nothing here asks you to.

What macOS *does* still ship, at `/bin/bash`, is genuinely relevant: bash 3.2, frozen since 2007 — the last version released before the GPLv3 switch. It's also missing features several later chapters rely on. Here's how to get a real, current bash installed alongside it, start to finish.

Installing a modern bash via Homebrew

1. Check whether Homebrew is already installed: `brew --version`. If that fails, install it with the official command from `brew.sh`:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. Install bash: `brew install bash`.
3. Confirm it worked: `bash --version` should now report 5.x, not 3.2.
4. Confirm it's the one `env` will find: `which bash` should print a Homebrew path (`/opt/homebrew/bin/bash` on Apple Silicon, `/usr/local/bin/bash` on Intel) — Homebrew adds its own directory near the front of `PATH` automatically during install.

NOTE

That last step is the entire reason Chapter 4 settled on `#!/usr/bin/env bash` instead of `#!/bin/bash`: once Homebrew's bash is first on `PATH`, every script here automatically picks it up — no shebang edits, no changing your login shell, nothing further to configure.

5.3 Getting a Modern Bash on Windows: WSL

Chapter 1 mentioned installing WSL (Windows Subsystem for Linux) without walking through it. Here's the full, current process:

1. Open PowerShell **as Administrator** (right-click the Start menu → "Terminal (Admin)" or "Windows PowerShell (Admin)").
2. Run `wsl --install`. This installs WSL itself and, by default, a full Ubuntu system alongside it.
3. Restart your computer if prompted — this step is common on a first install.
4. On first launch, Ubuntu asks you to create a Linux username and password. This is separate from your Windows login and can be anything you like; you'll use it any time a command needs elevated permission inside WSL.
5. Confirm it worked: open a terminal (Chapter 1's table — Windows Terminal, set to Ubuntu) and run `bash --version`.

Unlike macOS, this usually needs no further installation step: Ubuntu ships a current bash by default, so the version `wsl --install` gives you is typically already modern enough for everything ahead.

5.4 Why This Book Targets Bash 4+, Deliberately

The version boundary here isn't arbitrary. Bash 4.0 (2009) introduced associative arrays (Chapter 11), case-insensitive globbing, and several other features later chapters rely on directly. macOS's frozen 3.2 doesn't have any of them — not a degraded version, an absent one. A script using `declare -A` (an associative array) simply fails outright on 3.2, with an error that doesn't obviously point at "your bash is too old."

Checking your version takes one command, and it's worth running now on whichever platform you're on, before Part III needs it:

```
$ bash --version
GNU bash, version 5.2.21(1)-release (what you want to see – 4.x or higher)
```

5.5 A Quick Comparison Table You'll Come Back To

Everything in this chapter, side by side:

Platform	Default interactive shell	Getting bash 4+
Linux (Ubuntu, Debian)	bash, usually already	Nothing to do — already modern
macOS	zsh, since Catalina	<code>brew install bash</code> (§5.2) — system bash 3.2 is never enough
Windows	bash, inside WSL	<code>wsl --install</code> (§5.3) — usually modern out of the box

None of this changes how you write a script — every example runs identically on all three once bash 4+ is actually reachable. This is purely a one-time setup concern, not something you'll think about again once it's done.

CHAPTER 5 — QUICK RECAP

- On Debian/Ubuntu, `/bin/sh` is **dash**, not bash — bash-only syntax like `[[ ]]` fails under it. Use `#!/usr/bin/env bash` when you mean bash.
- macOS's default interactive shell is **zsh** since Catalina — irrelevant here, since every script is invoked as bash explicitly.
- macOS's system bash (`/bin/bash`) is a frozen 3.2 — install a modern one with `brew install bash`, then `#!/usr/bin/env bash` finds it automatically.
- Windows needs WSL (`wsl --install` in an admin PowerShell) — Ubuntu's bundled bash is usually already modern enough afterward.
- Requires **bash 4+** specifically because of features (associative arrays and more) that simply don't exist on 3.2. Check with `bash --version`.

That's the first five chapters.

Twenty-six more chapters pick up right where this sample leaves off — from arrays and parameter expansions through defensive scripting, real-world automation, testing with Bats, ShellCheck, security for privileged scripts, and a full quick-reference cheat sheet.

STILL IN THE FULL BOOK

- **Part II.** Bash Foundations, continued — Chapters 6–10
- **Part III.** Data, Text & Expansion
- **Part IV.** Writing Scripts That Don't Break
- **Part V.** Real-World Automation
- **Part VI.** Professional Practice

GET THE FULL BOOK

Available now at SplitZrate.com — same practical, no-fluff style as the five chapters you just read, all 31 chapters across six Parts, 135 pages total.