

way of the hax0r
n00b to console punk

Chapter 10

Make the Shell Yours

Every session so far started with the shell's defaults, plus whatever your operating system or a previous installer put in place. You have already seen settings worth keeping: an `EDITOR` variable, a `PATH` entry, a key binding, a prompt that shows the directory. Typing them at every prompt is not an option. The shell reads a **startup file** when it starts, and that file is where your version of the shell lives.

The concepts in this chapter are:

- **startup files** and **profiles**, and which one a given shell reads;
- **aliases** and **functions**, two ways to name a command of your own;
- the **prompt** as a setting you control;
- **history** settings: how much is kept and what is skipped;
- **completion** and **key bindings** as configuration;
- **recovery**: starting a shell without its configuration when a change breaks it.

Every change in this chapter is a text edit to a file in your home directory. Make one change at a time, open a new tab to test it, and keep the old tab open until the new one works.

Startup Files

A startup file is a script in the shell's own language that the shell runs before showing the first prompt. Anything you could type at the prompt can go in it, and its effects last for the session. Because the file runs every time, a mistake in it affects every new shell, which is why the last section of this chapter exists.

Which file runs depends on the shell and on how it was started. The two distinctions that matter are **interactive** versus non-interactive, and **login** versus non-login. A terminal tab is interactive. A script is not. A login shell is the first shell of a session: the one a text console or an SSH connection gives you, and, on macOS, every Terminal tab. Linux terminal tabs are usually non-login shells.

Shell	Interactive non-login	Login	Notes
Bash	<code>~/.bashrc</code>	<code>~/.bash_profile</code> , or <code>~/.profile</code> if that does not exist	Most <code>.bash_profile</code> files just run <code>.bashrc</code>

Shell	Interactive non-login	Login	Notes
zsh	<code>~/.zshrc</code>	<code>~/.zprofile</code> , then <code>~/.zshrc</code> , then <code>~/.zlogin</code>	<code>~/.zshenv</code> runs for every zsh, including scripts
PowerShell	<code>\$PROFILE</code>	same	Four files for user and host scopes
Command Prompt	none by default	none	An AutoRun registry value can name one

Put interactive settings such as aliases, prompts, and key bindings in `~/.bashrc` or `~/.zshrc`. Put environment variables that every program needs, such as `PATH` additions, in the login file so that they are set once and inherited. [1], [2]

Bash and zsh

Open the file for your shell in an editor and add a line at the end:

```
$ nano ~/.bashrc
```

For zsh, the file is `~/.zshrc`. Add `export EDITOR=nano`, save, and then load the file into the current shell to test it without opening a new tab:

```
$ source ~/.bashrc
$ echo "$EDITOR"
nano
```

`source`, also written as a single dot, runs a file in the current shell so that its variables and functions stay. Running the file as a program would set them in a child shell and lose them. [3]

If `~/.bash_profile` exists on a Linux machine and `~/.bashrc` changes do not appear in login shells, add these lines to `~/.bash_profile`:

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

macOS Terminal starts login shells, so on a Mac that file matters for every tab.

PowerShell

`$PROFILE` holds the path of the current user's profile for the current host, and it has four properties for the other scopes:

```
$ $PROFILE
$ $PROFILE | Select-Object *
$ Test-Path $PROFILE
```

The file does not exist until you create it:

```
$ New-Item -Path $PROFILE -ItemType File -Force
$ notepad $PROFILE
```

-Force creates the missing directories too. Use code `$PROFILE` or `nano $PROFILE` if you prefer. Add `$env:EDITOR = 'code --wait'`, save, and reload:

```
$ . $PROFILE
$ $env:EDITOR
```

The leading dot is **dot sourcing**, PowerShell's source. [4]

Windows PowerShell 5.1 and PowerShell 7 have separate profile files, in `WindowsPowerShell` and `PowerShell` folders under Documents on Windows. A profile written for one is not read by the other, and a profile that uses PowerShell 7 features fails to load in 5.1.



Trap: A profile is a script, so the execution policy from Chapter 8 applies. Under the default Restricted policy on Windows, PowerShell starts with an error saying the profile cannot be loaded, and nothing in it takes effect. Set the policy for your user to `RemoteSigned` first.

Command Prompt

Command Prompt reads no file at startup unless the `AutoRun` value under `HKCU\Software\Microsoft\Command Processor` in the registry names one. That value is a command line, typically `%USERPROFILE%\autorun.cmd`, and the script it names can set variables, define macros, and change the prompt. A simpler route is a Windows Terminal profile whose command line is `cmd.exe /k "%USERPROFILE%\autorun.cmd"`, which runs the script and keeps the window open. [5]

Aliases and Functions

An **alias** is a short name that the shell replaces with a longer command before running it. A **function** is a named block of shell code that can take arguments and contain logic. Use an alias for a fixed abbreviation and a function for anything that needs an argument in the middle.

Bash/zsh

```
$ alias ll='ls -aF'
$ alias gs='git status'
$ ll
$ alias
$ unalias gs
```

`alias` with no arguments lists them all. An alias is plain text substitution: `ll ~/Downloads` becomes `ls -aF ~/Downloads`. Arguments always go on the end, which is the limit of aliases. [6]

A function takes arguments as `$1`, `$2`, and so on:

```
mkcd() {
    mkdir -p "$1" && cd "$1"
```

```
}
```

Add that to your startup file, reload, and `mkcd projects/new` creates the directory and enters it. type `mkcd` reports it as a function, and `unset -f mkcd` removes it. Chapter 11 develops functions further. [7]

PowerShell

```
$ Set-Alias -Name ll -Value Get-ChildItem
$ Get-Alias ll
$ Remove-Alias ll
```

A PowerShell alias maps a name to a command, and only to a command: it cannot include parameters. `Set-Alias ll 'Get-ChildItem -Force'` is accepted, but running `ll` then fails, because no command is named `Get-ChildItem -Force`. Remove such an alias with `Remove-Alias`; an alias also takes precedence over a function of the same name, so the broken alias would hide a working function. For anything with parameters, write a function: [8]

```
function ll { Get-ChildItem -Force @args }
function mkcd {
    param([string]$Path)
    New-Item -ItemType Directory -Path $Path -Force | Out-Null
    Set-Location -LiteralPath $Path
}
```

`@args` passes along whatever arguments the function received. Put these in `$PROFILE` and dot-source it. [9]

Command Prompt

`doskey` defines **macros**, which are Command Prompt's aliases:

```
$ doskey ll=dir /a $*
$ doskey /macros
```

`$*` stands for the arguments. Macros last for the session; to keep them, put the `doskey` lines in the `AutoRun` script. [10]

The Prompt

The prompt is a string the shell builds from a template each time it needs one. Changing it is the most visible customization and one of the most useful: a prompt that shows the current directory answers the question Chapter 3 kept asking.

Bash

`PS1` holds the template. Backslash sequences stand for pieces of information:

```
$ PS1='\u@\h:\w\$ '
student@laptop:~/projects$
```

\u is the username, \h the hostname, \w the current directory with ~ for home, and \\$ a \$ that becomes # for root. \W gives only the last directory name, and \n starts a new line. Colors use escape sequences wrapped in \[and \] so that the shell can measure the prompt's length correctly. [11]

zsh

zsh uses PROMPT with percent sequences:

```
$ PROMPT='%n@m:%~%# '
```

%n is the username, %m the hostname, %~ the directory, and %# the % or # marker. %F{blue} and %f turn color on and off, and %? is the last exit status. [12]

PowerShell

The prompt is a function named prompt, and redefining it changes the prompt:

```
function prompt {
    "PS $(Split-Path -Leaf (Get-Location))> "
}
```

The function can run any code, which is how prompts show Git branches or timing. Keep it fast, because it runs before every command. [13]

Command Prompt

prompt \$P\$G is the default: the path and a >. \$D adds the date, \$T the time, and \$_ a new line. Put a prompt line in the AutoRun script to keep it. [14]

History Settings

Chapter 4 used history; these settings decide what it keeps.

Bash:

```
HISTSIZE=10000
HISTFILESIZE=20000
HISTCONTROL=ignoreboth
shopt -s histappend
```

HISTSIZE is how many commands the shell remembers, HISTFILESIZE how many the file keeps, and ignoreboth skips duplicates and commands that start with a space. histappend makes each shell add to the file instead of replacing it, so two tabs do not erase each other's history. [15], [16]

zsh:

```
HISTFILE=~/.zsh_history
HISTSIZE=10000
SAVEHIST=10000
setopt SHARE_HISTORY HIST_IGNORE_DUPS HIST_IGNORE_SPACE
```

zsh keeps no history file until HISTFILE is set. SHARE_HISTORY makes commands from one tab visible in another immediately. [17], [18]

PowerShell:

```
Set-PSReadLineOption -HistoryNoDuplicates
Set-PSReadLineOption -HistorySearchCursorMovesToEnd
```

(Get-PSReadLineOption).HistorySavePath shows where PSReadLine keeps its file. -MaximumHistoryCount limits it, and -AddToHistoryHandler lets a script block decide which commands to keep, which people use to skip commands containing passwords. [19]

Command Prompt keeps no history between sessions, and there is no setting to change that.

Completion and Key Bindings

Bash: Readline reads ~/.inputrc. Two lines that most people want:

```
set completion-ignore-case on
"\e[A": history-search-backward
"\e[B": history-search-forward
```

The first makes Tab ignore case. The other two make Up and Down Arrow search history for commands that start with what you have typed, instead of walking through everything. bind -f ~/.inputrc reloads it, and bind -p shows the result. Install the bash-completion package for completion of options and arguments, and check that your .bashrc sources it. [20]

zsh: The completion system needs to be turned on:

```
autoload -Uz compinit && compinit
bindkey -e
bindkey '^[[A' history-search-backward
bindkey '^[[B' history-search-forward
```

bindkey -e chooses Emacs keys explicitly, which matters after the EDITOR surprise in Chapter 4. [21], [22]

PowerShell:

```
Set-PSReadLineOption -EditMode Emacs
Set-PSReadLineKeyHandler -Key Ctrl+Spacebar -Function MenuComplete
Set-PSReadLineKeyHandler -Key UpArrow -Function HistorySearchBackward
Set-PSReadLineKeyHandler -Key DownArrow -Function HistorySearchForward
Set-PSReadLineOption -PredictionSource History
```

The first line picks an edit mode regardless of platform. The next three bind functions to keys; Get-PSReadLineKeyHandler lists the function names you can use. The last line turns on inline suggestions from history. [19], [23]

Completion for a specific command's arguments comes from Register-ArgumentCompleter, which tools such as Git and the .NET CLI provide scripts for. [24]

When the Configuration Breaks

A startup file that fails leaves symptoms: an error message before the first prompt, a missing prompt, a shell that exits immediately, or a shell that hangs. The fix is the same each time: start a shell that ignores the file, then repair the file.

Shell	Start without configuration
Bash	<code>bash --norc --noprofile</code>
zsh	<code>zsh -f</code>
PowerShell	<code>pwsh -NoProfile</code> or <code>powershell -NoProfile</code>
Command Prompt	<code>cmd /d</code>

Run one of those from a working shell, or set it as the command line of a spare terminal profile so that you always have a way in. Then open the startup file in an editor and look at the last change you made. If you cannot remember, comment out the second half of the file with `#`, test, and narrow it down. [5], [25]

Two habits prevent most of this. Test each addition with `source` or dot-sourcing in the current tab before opening a new one. And keep your startup files in a Git repository, which Chapter 14 covers, so that `git diff` shows exactly what changed.



Tip: Add `set -x` near the top of a Bash startup file temporarily to print every line as it runs. The last line printed before the shell hangs or fails is the one to look at. Remove it afterward.

Key Points

- Interactive settings go in `~/.bashrc`, `~/.zshrc`, or `$PROFILE`; environment variables for every program go in the login file.
- `source file` and `. file` run a file in the current shell so its settings stay.
- Aliases substitute text and take arguments only at the end; functions take arguments anywhere. PowerShell aliases cannot include parameters.
- The prompt is `PS1` in Bash, `PROMPT` in zsh, the `prompt` function in PowerShell, and the `prompt` command in Command Prompt.
- History size, duplicate handling, and sharing between tabs are settings in the startup file; Command Prompt has none.
- `~/.inputrc`, `bindkey`, and `Set-PSReadLineKeyHandler` change key bindings; zsh needs `compinit` for completion.
- `bash --norc`, `zsh -f`, `pwsh -NoProfile`, and `cmd /d` start a shell without its configuration for recovery.

Level Up

n00bs finish the chapter here. You can keep a setting, an alias, and a key binding across sessions and recover when a change breaks the shell. Hackers treat their configuration as a project.

Look at a prompt framework: **Starship** works in every shell in this book, **Oh My Posh** targets PowerShell first, and **Oh My Zsh** is a whole plugin system. Install one and read what it adds to your startup file. Which of its features would you have written yourself, and what does it cost in startup time? Measure with `time bash -ic exit` or `Measure-Command { pwsh -NoProfile -c exit }` against your normal startup.

Start a **dotfiles** repository: put your startup files in Git and symlink them into place, or look at tools such as **chezmoi** and **stow**. Explain how you would set up a new machine from it.

Read about **direnv**, which loads environment variables when you enter a directory, and about PowerShell's `$PSDefaultParameterValues`, which changes a cmdlet's defaults. When is a per-directory setting better than a global one?

Try `PSReadLine's -EditMode Vi` and `Readline's set editing-mode vi` for a week, and write down which keys you kept.

Exercises

1. **What does this do?** Predict what `alias rm='rm -i'` changes about `rm -rf build`, then explain why some people consider that alias dangerous.
2. **What does this do?** Predict the output of `PS1='\W \$ '` in a Bash session sitting in `~/projects/game`.
3. **What is the problem?** A user adds `export PATH=/opt/tool/bin` to `~/.bashrc` and every command afterward reports “command not found”. Explain what happened and how to recover from that terminal.
4. **What is the problem?** A PowerShell profile works in PowerShell 7 but Windows PowerShell ignores it. Explain where each shell looks.
5. **Why is it like this?** `Set-Alias` cannot include parameters, but Bash `alias` can. Explain what each shell substitutes and why PowerShell offers functions instead.
6. **Why is it like this?** A `PATH` change in `~/.bashrc` is invisible to a graphical editor launched from the desktop. Explain which processes read that file.
7. **How do you** make Up Arrow search history by prefix in Bash, zsh, and PowerShell?
8. **How do you** find out which startup file set an alias that you did not write?

References

- [1] GNU Project, “Bash reference manual: Bash startup files.” Available: https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html
- [2] Zsh Development Group, “The z shell manual: files.” Available: <https://zsh.sourceforge.io/Doc/Release/Files.html>
- [3] GNU Project, “Bash reference manual: Bourne shell builtins (source).” Available: https://www.gnu.org/software/bash/manual/html_node/Bourne-Shell-Builtins.html
- [4] Microsoft, “About_profiles.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_profiles
- [5] Microsoft, “Windows commands: cmd.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cmd>
- [6] GNU Project, “Bash reference manual: aliases.” Available: https://www.gnu.org/software/bash/manual/html_node/Aliases.html
- [7] GNU Project, “Bash reference manual: Shell functions.” Available: https://www.gnu.org/software/bash/manual/html_node/Shell-Functions.html
- [8] Microsoft, “Using aliases.” Available: <https://learn.microsoft.com/en-us/powershell/scripting/learn/shell/using-aliases>
- [9] Microsoft, “About_functions.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions
- [10] Microsoft, “Windows commands: doskey.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/doskey>
- [11] GNU Project, “Bash reference manual: Controlling the prompt.” Available: https://www.gnu.org/software/bash/manual/html_node/Controlling-the-Prompt.html
- [12] Zsh Development Group, “The z shell manual: Prompt expansion.” Available: <https://zsh.sourceforge.io/Doc/Release/Prompt-Expansion.html>
- [13] Microsoft, “About_prompts.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_prompts
- [14] Microsoft, “Windows commands: prompt.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/prompt>
- [15] GNU Project, “Bash reference manual: Bash history facilities.” Available: https://www.gnu.org/software/bash/manual/html_node/Bash-History-Facilities.html
- [16] GNU Project, “Bash reference manual: The shopt builtin.” Available: https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html

- [17] Zsh Developers, “The z shell manual: parameters.” Available: <https://zsh.sourceforge.io/Doc/Release/Parameters.html>
- [18] Zsh Development Group, “The z shell manual: options.” Available: <https://zsh.sourceforge.io/Doc/Release/Options.html>
- [19] Microsoft, “Set-PSReadLineOption.” Available: <https://learn.microsoft.com/en-us/powershell/module/psreadline/set-psreadlineoption>
- [20] GNU Project, “Bash reference manual: Readline init file.” Available: https://www.gnu.org/software/bash/manual/html_node/Readline-Init-File.html
- [21] Zsh Development Group, “The z shell manual: Completion system.” Available: <https://zsh.sourceforge.io/Doc/Release/Completion-System.html>
- [22] Zsh Development Group, “The z shell manual: Zsh line editor.” Available: <https://zsh.sourceforge.io/Doc/Release/Zsh-Line-Editor.html>
- [23] Microsoft, “Set-PSReadLineKeyHandler.” Available: <https://learn.microsoft.com/en-us/powershell/module/psreadline/set-psreadlinekeyhandler>
- [24] Microsoft, “Register-ArgumentCompleter.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/register-argumentcompleter>
- [25] GNU Project, “Bash reference manual: Invoking bash.” Available: https://www.gnu.org/software/bash/manual/html_node/Invoking-Bash.html