

way of the hax0r
n00b to console punk

Chapter 7

Make Commands Work Together

Each command so far did one job and printed the result. The shell's real power is connecting those jobs: sending one command's output into another, saving output to a file, and running a second command only if the first succeeded. A chain of small tools built this way does work that would otherwise need a program written specially for it.

The concepts in this chapter are:

- the three **standard streams**: input, output, and error;
- **redirection**, which connects a stream to a file;
- **pipes**, which connect one command's output to another's input;
- **exit status**, a command's report of success or failure;
- **conditional execution** with `&&`, `||`, and `;;`;
- **text filters**, the small UNIX tools built to sit in a pipeline;
- PowerShell's **object pipeline**, and how it differs from a text pipeline.

The examples read the practice set from Chapter 6, so recreate it if you removed it. Redirection can overwrite files, and the text says so where it applies.

Three Streams

Every program starts with three connections, called **standard streams**:

- **standard input** (stdin), where it reads from, usually the keyboard;
- **standard output** (stdout), where it writes results, usually the terminal;
- **standard error** (stderr), where it writes error messages, also usually the terminal.

Each has a number: 0, 1, and 2. The terminal shows output and errors mixed together, which hides the fact that they are separate streams. The separation matters as soon as you redirect one of them: results go to a file while errors still reach your screen, or the other way round.

The shell sets up those connections before the program starts. **Redirection** points a stream at a file. A **pipe** points one program's standard output at another program's standard input. The programs themselves do not know or care; `sort` reads its input the same way whether it comes from the keyboard, a file, or another program.

Redirect to Files

Bash/zsh:

```
$ sort notes.txt > sorted.txt
$ echo 'epsilon' >> notes.txt
$ wc -l < notes.txt
5
$ ls notes.txt nowhere 2> errors.txt
notes.txt
$ ls notes.txt nowhere > all.txt 2>&1
```

> sends standard output to a file, replacing the file's contents; >> appends. < reads standard input from a file. 2> redirects standard error, so the first `ls` shows only its success on screen and puts the error in `errors.txt`. 2>&1 sends standard error to wherever standard output is going at that moment, so the last command collects both in `all.txt`. [1]

The order matters: `2>&1 > all.txt` sends errors to the terminal, because at the time `2>&1` is processed, standard output still pointed there.

PowerShell:

```
$ Get-Content notes.txt | Sort-Object > sorted.txt
$ 'epsilon' >> notes.txt
$ Get-ChildItem notes.txt, nowhere 2> errors.txt
$ Get-ChildItem notes.txt, nowhere *> all.txt
```

>, >>, and 2> work as in Bash. PowerShell has more than three streams: besides output and error there are warning, verbose, debug, and information streams, numbered 3 to 6. *> redirects all of them, and 2>&1 merges errors into output as in Bash. There is no <; commands read files with `Get-Content` instead. [2], [3]

Command Prompt:

```
$ sort notes.txt > sorted.txt
$ echo epsilon >> notes.txt
$ dir notes.txt nowhere 2> errors.txt
$ dir notes.txt nowhere > all.txt 2>&1
```

The syntax matches Bash for these forms, including the ordering rule for 2>&1. < file supplies standard input. [4]



Trap: > truncates the target before the command runs. `sort notes.txt > notes.txt` empties `notes.txt` first and then sorts nothing. Write to a new file and rename it afterward.

/dev/null on UNIX and \$null in PowerShell are places to send output you do not want to see. `grep -r pattern . 2>/dev/null` hides permission errors; Command Prompt uses nul.

Pipe One Command into Another

A **pipeline** joins commands with `|`. The shell runs them at the same time and streams the first one's output into the second one's input.

Bash/zsh:

```
$ cat notes.txt | sort
$ sort notes.txt | uniq -c
$ grep -r main . | wc -l
1
```

The first pipeline is a habit to break: `sort notes.txt` reads the file itself, so `cat` adds nothing. The second sorts and then counts repeated lines; `uniq` only sees repeats that are adjacent, which is why `sort` comes first. The third counts matching lines across a tree. [5]

Longer pipelines are read left to right, one stage at a time. Build them the same way: run the first command, check its output, add `| next-command`, and check again.

PowerShell:

```
$ Get-Content notes.txt | Sort-Object
$ Get-ChildItem -Recurse -File | Select-String main | Measure-Object
```

The syntax is the same, but what flows through the pipe is different, and the next section is about that difference.

Command Prompt:

```
$ type notes.txt | sort
$ dir /b | findstr txt
```

Command Prompt pipes text. Its set of filters is small: `sort`, `find`, `findstr`, and more. [6], [7]

Text Pipelines and Object Pipelines

In Bash, zsh, and Command Prompt, a pipe carries **bytes**, usually lines of text. Every tool must parse the text it receives and print text for the next one. That is why `ls -l | sort -k5 -n` needs `sort` to be told which column holds the size and that it is a number: the pipe carries a picture of the listing, not the listing itself.

In PowerShell, a pipe carries **objects**. `Get-ChildItem` sends `FileInfo` objects with properties such as `Name`, `Length`, and `LastWriteTime`, and the next cmdlet reads those properties by name:

```
$ Get-ChildItem -Recurse -File | Where-Object Length -gt 20 | Sort-Object Length -Descending |
```

`Where-Object` filters by a property, `Sort-Object` sorts by one, and `Select-Object` keeps the properties you want. No column numbers and no parsing. `ForEach-Object` runs a block for each object, with `$_` standing for the current one: [8], [9], [10], [11], [12]

```
$ Get-ChildItem -File | ForEach-Object { $_.Name.ToUpper() }
```

To see what properties an object has, pipe it to `Get-Member`:

```
$ Get-Item notes.txt | Get-Member -MemberType Property
```

That command is how you discover what `Where-Object` and `Sort-Object` can use, and it is worth more than any table of cmdlets. [13]

Text still appears in PowerShell pipelines. `Get-Content` sends strings, one per line, and an external program such as `git` or `grep` sends strings too. When you need text from objects, `Out-String` makes it; when you need objects from text, `ConvertFrom-Csv` and `ConvertFrom-Json` parse structured formats.



Trap: `Format-Table` and `Format-List` produce formatting objects, not the original data. Put them last in a pipeline. `Get-ChildItem | Format-Table | Sort-Object Length` sorts nothing useful.

The table shows the same intent in both models:

Intent	Bash/zsh text pipeline	PowerShell object pipeline
Keep matching lines or objects	<code>grep pattern</code>	<code>Where-Object { \$_ -match 'pattern' }</code>
Sort	<code>sort, sort -n</code>	<code>Sort-Object, Sort-Object Property</code>
Take the first few	<code>head -n 3</code>	<code>Select-Object -First 3</code>
Count	<code>wc -l</code>	<code>Measure-Object</code>
Pick a column or property	<code>cut -d, -f2, awk '{print \$2}'</code>	<code>Select-Object Property</code>
Remove duplicates	<code>sort \ uniq</code>	<code>Sort-Object -Unique</code>
Do something with each item	<code>xargs, while read loop</code>	<code>ForEach-Object</code>

Text Filters

A **filter** reads standard input, transforms it, and writes standard output. The classic UNIX filters each do one thing, and they combine well because of the streams above.

Filter	Job	Example
<code>sort</code>	Sort lines; <code>-n</code> numerically, <code>-r</code> reversed, <code>-u</code> unique	<code>sort -n sizes.txt</code>
<code>uniq</code>	Collapse adjacent duplicate lines; <code>-c</code> counts them	<code>sort words \ uniq -c</code>
<code>cut</code>	Pick fields by delimiter or character position	<code>cut -d, -f2 data.csv</code>
<code>tr</code>	Translate or delete characters	<code>tr a-z A-Z</code>
<code>head, tail</code>	First or last lines	<code>tail -n 5</code>
<code>grep</code>	Keep matching lines	<code>grep -v '^#'</code>
<code>wc</code>	Count lines, words, bytes	<code>wc -l</code>

Filter	Job	Example
tee	Write input to a file and also pass it on	make \ tee build.log

Try one that combines several:

```
$ tr a-z A-Z < notes.txt | sort | uniq | head -n 3
ALPHA
BETA GAMMA
DELTA
```

Each stage is easy to test alone, and that is the design. [14], [15], [16], [17], [18]

sed and awk are the two heavier filters: sed edits lines with regular expressions, and awk splits lines into fields and can compute with them. sed 's/gamma/GAMMA/' notes.txt replaces the first gamma on each line, and awk '{print \$1}' notes.txt prints the first word of each. Both deserve time later; recognize them now. [19], [20]

Exit Status

Every command ends by reporting a number called its **exit status**. Zero means success. Any other value means failure, and the specific number sometimes says why. The shell records the status of the last command so that you can inspect it and act on it.

Bash/zsh:

```
$ grep zzz notes.txt
$ echo $?
1
$ grep alpha notes.txt
alpha
$ echo $?
0
```

\$? holds the last exit status. grep returns 1 when nothing matched, which is a report and not an error, and 2 when something went wrong such as a missing file. In a pipeline, \$? reports the last command's status by default; set -o pipefail makes a failure anywhere in the pipeline count. [21], [22]

PowerShell:

```
$ grep zzz notes.txt
$ $LASTEXITCODE
1
$ Select-String zzz notes.txt
$ $?
True
```

PowerShell keeps two records. `$LASTEXITCODE` is the exit status of the last external program. `$?` is True if the last command ran without error, and a cmdlet that finds nothing has not errored, so `$?` stays True after `Select-String` finds no match. [23]

Command Prompt:

```
$ findstr zzz notes.txt
$ echo %ERRORLEVEL%
1
```

`%ERRORLEVEL%` holds the last exit status. [24]

Run the Next Command Only If

Three operators sequence commands based on exit status.

- `first && second` runs second only if first succeeded.
- `first || second` runs second only if first failed.
- `first ; second` runs both regardless, in Bash, zsh, and PowerShell. Command Prompt uses `&` for this.

Bash/zsh:

```
$ mkdir build && cd build
$ grep zzz notes.txt || echo 'not found'
not found
$ cd ..
```

The first line enters `build` only if it was created; without `&&`, a failed `mkdir` would leave you running the rest somewhere unexpected. [25]

PowerShell:

```
$ New-Item -ItemType Directory build && Set-Location build
$ grep zzz notes.txt || Write-Output 'not found'
```

`&&` and `||` were added in PowerShell 7. Windows PowerShell 5.1 does not have them, and a script that uses them fails to parse there; use `if ($?) { ... }` instead. [26]

Command Prompt:

```
$ mkdir build && cd build
$ findstr zzz notes.txt || echo not found
```

Command Prompt has had `&&` and `||` for decades, with `&` as the unconditional separator. [24]

Key Points

- Every program has standard input, output, and error; the shell connects them to files with `>`, `>>`, `<`, and `2>`, and `2>&1` merges errors into output.

- | connects one command's output to another's input, and a pipeline is built and tested one stage at a time.
- Bash, zsh, and Command Prompt pipe text; PowerShell pipes objects with properties, which Get-Member reveals.
- sort, uniq, cut, tr, grep, wc, and tee are filters that combine; Where-Object, Sort-Object, Select-Object, and ForEach-Object are their object counterparts.
- Zero exit status means success. Read it from \$?, \$LASTEXITCODE, or %ERRORLEVEL%.
- && and || run the next command based on the previous status; PowerShell gained them in version 7.

Level Up

n00bs finish the chapter here. You can redirect, pipe, read an exit status, and chain commands on success or failure. Hackers make the pipeline their main tool.

Learn xargs and find -exec, then read about -print0 and xargs -0 and explain the filename problem they solve. Compare with ForEach-Object -Parallel in PowerShell 7.

Explore **process substitution**, <(command), in Bash and zsh: diff <(sort a.txt) <(sort b.txt) compares two sorted lists without temporary files. Explain what the shell creates to make that work.

Spend an hour with awk: sum a column, print lines longer than 80 characters, and count words. Then do the same three tasks in PowerShell with objects, and compare the two approaches honestly.

Investigate how PowerShell binds pipeline input to parameters, using Get-Help Get-Item -Full and looking for "Accept pipeline input". Why does "notes.txt" | Get-Item work but "notes.txt" | Get-Content does not?

Read about set -o pipefail and set -e, and about \$PSNativeCommandUseErrorActionPreference in PowerShell 7.4 and later. When would each one make a script safer, and when would it make it stop for the wrong reason?

Exercises

1. **What does this do?** Predict the output of `printf 'b\na\nb\n' | sort | uniq -c`, then run it. Then remove sort and explain the change.
2. **What does this do?** Predict what `Get-ChildItem | Sort-Object Length -Descending | Select-Object -First 1 -ExpandProperty Name` prints in your practice directory.
3. **What is the problem?** `ls -l | sort -k5 -n` sorts by size, but adding `| head -n 1` prints the total line instead of a file. Explain where that line came from.
4. **What is the problem?** A PowerShell 5.1 user pastes `git pull && git push` from a tutorial and gets a parse error. Explain why and rewrite it for 5.1.
5. **Why is it like this?** `grep` returns exit status 1 when it finds nothing, even though nothing went wrong. Explain why that is useful in a script.
6. **Why is it like this?** `command 2>&1 > file` sends errors to the terminal, not the file. Explain the order in which the shell applies the two redirections.
7. **How do you** count how many files under the current directory contain the word `main`, in Bash and in PowerShell?

8. **How do you** save a command's output to a file and still watch it scroll past, in Bash and in PowerShell?

References

- [1] GNU Project, “Bash reference manual: redirections.” Available: https://www.gnu.org/software/bash/manual/html_node/Redirections.html
- [2] Microsoft, “About_redirection.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_redirection
- [3] Microsoft, “About_output_streams.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_output_streams
- [4] Microsoft, “Windows commands: Using command redirection operators.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/redirection>
- [5] GNU Project, “Bash reference manual: pipelines.” Available: https://www.gnu.org/software/bash/manual/html_node/Pipelines.html
- [6] Microsoft, “Windows commands: sort.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/sort>
- [7] Microsoft, “Windows commands: findstr.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/findstr>
- [8] Microsoft, “About_pipelines.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pipelines
- [9] Microsoft, “Where-object.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object>
- [10] Microsoft, “Sort-object.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/sort-object>
- [11] Microsoft, “Select-object.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-object>
- [12] Microsoft, “ForEach-object.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/foreach-object>
- [13] Microsoft, “Get-member.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-member>
- [14] GNU Project, “GNU coreutils manual: Sort: Sort text files.” Available: https://www.gnu.org/software/coreutils/manual/html_node/sort-invocation.html
- [15] GNU Project, “GNU coreutils manual: Uniq: Uniquify files.” Available: https://www.gnu.org/software/coreutils/manual/html_node/uniq-invocation.html
- [16] GNU Project, “GNU coreutils manual: Cut: Print selected parts of lines.” Available: https://www.gnu.org/software/coreutils/manual/html_node/cut-invocation.html

- [17] GNU Project, “GNU coreutils manual: Tr: Translate, squeeze, and/or delete characters.” Available: https://www.gnu.org/software/coreutils/manual/html_node/tr-invocation.html
- [18] GNU Project, “GNU coreutils manual: Tee: Redirect output to multiple files or processes.” Available: https://www.gnu.org/software/coreutils/manual/html_node/tee-invocation.html
- [19] GNU Project, “GNU sed manual.” Available: <https://www.gnu.org/software/sed/manual/sed.html>
- [20] GNU Project, “The GNU awk user’s guide.” Available: <https://www.gnu.org/software/gawk/manual/gawk.html>
- [21] GNU Project, “Bash reference manual: Exit status.” Available: https://www.gnu.org/software/bash/manual/html_node/Exit-Status.html
- [22] GNU Project, “GNU grep manual.” Available: <https://www.gnu.org/software/grep/manual/grep.html>
- [23] Microsoft, “About_automatic_variables.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables
- [24] Microsoft, “Windows commands: cmd.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cmd>
- [25] GNU Project, “Bash reference manual: Lists of commands.” Available: https://www.gnu.org/software/bash/manual/html_node/Lists.html
- [26] Microsoft, “About_pipeline_chain_operators.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pipeline_chain_operators