3

Using the Command Line

If you're a casual Mac user, or even if you're a hard-core Linux or Unix user, there are a few things about Mac OS X and the particular flavor of Unix under its candylike shell that might catch you off guard. Files and folders behave in rather different ways when you're addressing them with textual commands than when you're shoving them around with your mouse. Not only do they look different, they act different, too. You might even say they "think different."

The shell, which is what we call the command-line environment displayed by the Terminal application, is an austere and cryptic piece of software—about as un-Mac-like as it can possibly get. By the end of this book, you'll have found all kinds of uses for it—tricks that weren't otherwise possible using the graphical Aqua interface. But there's a steep learning curve, particularly for readers who have never dabbled in Unix before, and there are a few things you're going to have to know about how your files work in the shell before you can really start ordering them around.

NOTE: This will be discussed in more detail in Chapter 4, "Basic Unix Commands," but you should be aware that every Unix command is fully documented within the command line using the `man` ("manual") command. Type **`man`** ***`command`*** to learn more about any command you've heard about.

Everything Is a File

Your Mac is designed primarily to show you your documents, folders, applications, and other items in neatly ordered windows, with pretty icons next to them to help you differentiate them based on their type. You can open Finder windows that show you each item's Kind in a column, distinguishing your Photoshop images from your Word documents and your folders and applications. Mac OS X even has "bundles," which are special folders full of executables and other items masquerading as single monolithic files in the Finder, which you'll learn more about in Chapter 5, "Using the Finder." At the graphical level, your Mac is full of all kinds of items that each get their own unique look and descriptive vocabulary.

Not much of that matters at the command-line level. Your shell doesn't see a folder differently from how it sees a Word document; they're both just "streams of bits with names" as far as it's concerned, and in its 1970s-era worldview that's all that matters. The only thing distinguishing a folder (or *directory*) from a file is that the bits in it describe links to other files that the operating system should interpret as part of that folder, rather than the binary or textual data stream that make up a file's contents—but to Unix that's trivia. If you use the `ls` ("list") command in the shell to list the files

in a folder, you'll just get a list of names—no icons, no turn-down arrows, no clues to help tell you that some of the things you're looking at are files and some are folders, applications, or what-have-you. (There are some options you can give to the `ls` command to make it smarter about how it lists the items, as you'll see later; but that's a courtesy that Unix only grudgingly grants.)

In the Unix world, everything's a file, including such oddities as running processes and network connections and attached devices, and you interact with them all in pretty much the same way, using the same commands for everything (with a few exceptions, like the `mkdir` command). I point this out to make you aware that if you see the command-line examples in this book refer to "files," it means "files, folders, and any other discrete pieces of data." If a command makes a distinction between regular data files and other kinds of items, I'll say so; but otherwise, you can generally expect that a command will work the same on one kind of item as on another, because it'll see "files" with as little discrimination as Unix does.

File Types and Extensions

The Unix side of Mac OS X might not care about what makes one kind of file different from another, but the graphical side certainly does. The "kind" of a file, which you can view by selecting it and then choosing **File**, **Get Info**, is what determines what kind of icon it has in the Finder and, more importantly, what application it opens in when you double-click it. This is pretty basic stuff, and it's familiar to anyone who's used a Windows PC or Mac anytime in the past 20 years.

What you might not be familiar with is just *how* Mac OS X identifies a file's kind. In the old, pre-OS X days, files on the Mac had an invisible four-letter "Type" code, along with another four-letter "Creator" code, the combination of which told the system what application the file belonged to and what other apps could open it if they advertised themselves as being able to open, for example, "JPEG" pictures or "MooV" movie files. Because these codes were invisible, nobody had to deal with them or even know they were there, and—even better—nobody had to put up with those ugly "extensions" they'd seen on files in Windows or MS-DOS. Why should you have to name a file "Shopping List.txt" when you could just call it "Shopping List" and have the system *know* it was a text file because of its TEXT Type code?

Mac OS X brought an end to that happy and elegant time, to many users' (and my) chagrin. Now, instead of Type and Creator codes, files were identified using extensions, just like in Windows: `.txt` for text files, `.doc` for Microsoft Word documents, `.jpg` for JPEG pictures, and so on. On the face of it, this looks like a huge step backward for usability. But what it really was was a nod to reality; the world in 2001 was dominated by Windows, and that meant that every file on the Internet had extensions, so we might as well get used to it. But Mac users don't have to like it. And that's why extensions in Mac OS X, after some early rough edges were sanded off, are handled with arguably even more slickness and flexibility than Type and Creator codes were.

You can hide the extension on a file, on a per-file basis (unlike in Windows, where either all extensions are shown or only the unknown ones are, as dictated by a global setting). Better yet, the way you hide an

extension is by simply renaming the file: You click the filename, you put the cursor at the end, you backspace out the `.txt` or `.doc`, and it's gone, just as if the extension were any meaningless and disposable part of the filename. But it's not really gone: Do a Get Info on the file, and you'll find that the `.txt` or `.doc` is still there—the Name & Extension field shows the complete name, and the Hide Extension check box is checked. The system is similarly smart enough to figure out whether to hide or show the extension when you save a new file in TextEdit or Preview; if you specify the extension, it's shown, but if you don't, it's hidden.

Why is this useful? Why not just use Type codes like in the old days? Well, think about interoperability. If the filenames didn't have extensions, and you sent a text file or an MP3 song to someone using Windows, his computer wouldn't know what to do with it. Windows and Linux can't read Type and Creator codes, and those codes aren't included with files when transferred through popular Internet apps anyway. But if the extensions are there, and they're hidden only for the benefit of Mac users, then Windows and Linux users can still open the files using their favorite text editors or MP3 players, and Mac users can still look at pretty, extensionless filenames. Everybody wins!

TIP: Rather than the "Creator" of each individual file being stored in metadata, Mac OS X keeps a database of "opener apps" for known file types. The default opener for JPEG images, for example, is Preview. You can set individual files to open in other apps, though, and you can change the default opener of a given file type; in the **Get Info** window, open up the **Open with** panel to configure these behaviors.

Be aware that just because you don't see an extension on a filename in the Finder, that doesn't mean the extension isn't there. If you look at the file in the command-line shell, you'll see the whole filename, extension and all.

TIP: Unix has its own, entirely separate way of figuring out what kind of files you're looking at: It looks at the file's contents and makes an educated guess. This functionality isn't part of your shell or any universal system service, though—it's accomplished using the `file` command, which you can use like so:

```
Silver:~/Pictures btiemann$ file pvp.psd
pvp.psd: Adobe Photoshop Image
```

Maximum Filename Lengths

One of the benefits that Mac OS X brought to the Mac-using world was longer filenames. In the old Mac OS, 31 characters were all you had to work with; you didn't have to worry about extensions, but 31 was still too short, for instance, for naming an MP3 file according to its title, artist, and album. MS-DOS, if your memory is that long, was even worse: eight characters, all in caps, and a three-letter extension. How did we ever survive?

But now we have a full 255 characters to devote to any filename, and that includes spaces, quotes, apostrophes, and all kinds of other characters (with a few exceptions, as you'll see shortly). I don't care how long the title of your favorite MP3 is; you're not going to run out of letters to describe it in Mac OS X.

One thing to watch out for, though, is that when filenames get too long to be displayed comfortably, they start to wreak havoc on the mechanisms used to display them, both in the graphical and command-line levels. The Finder will shorten a displayed filename to a reasonable length and stick an ellipsis (…) in the middle to show you that there's more to the filename than what you see. But the Unix shell is less sophisticated and will dutifully print out the whole massive filename, even if it wraps four times in your 80-column-wide display and wrecks the format of your file listing. To keep your own sanity, to say nothing of good desktop hygiene, you should probably keep your filenames to around 30–40 characters at most. But that's just some motherly advice, not a requirement of the system.

Case Sensitivity and Case Preservation in Filenames

Where Mac OS X differs most visibly from other Unixes is in the way its filesystem (HFS+, for those of you keeping score) handles capitalization in filenames. Most Unix-style operating systems are *case sensitive*, meaning that a file called `File1.txt` is entirely distinct from one called `file1.txt`, and both can happily exist in the same folder. Linux or FreeBSD will see not the slightest similarity between those two files, no matter how much our human sensibilities might tell us that they're the same.

Mac OS X, like the classic Mac OS before it, is *not* case sensitive; it doesn't care whether you said `File1.txt` or

`file1.txt`. Only one of them can exist in a folder at the same time, and there's no ambiguity for either computers or humans in telling which file you meant. Even Unix commands like `ls` will work if you give them filenames to operate on that don't match the capitalization of the actual files (try it: **`ls /library`**).

NOTE: Because bash and other shells packaged with Mac OS X were developed outside Apple and without this kind of flexible case handling in mind, Tab completion won't work unless you use the correct capitalization. For instance, typing **`/lib`** and pressing Tab won't do anything, but **`/Lib`** followed by Tab will expand to **`/Library`**.

However, unlike some versions of Windows, Mac OS X is also *case preserving*. If you create a file called `file1.txt`, the system will keep it as `file1.txt`; it won't helpfully capitalize the first letter for you, it won't force the whole thing to uppercase or lowercase, and it won't lose track of the capitalization if you send the file through one application and then another, or up to a web server and back down again. Things stay the way you put them, but the system can generally figure out what you mean if you're less precise than it is. Unix purists who insist that the byte for "a" is as different from "A" as it is from "9" might grouse, but Mac OS X is just behaving the way humans do, isn't it?

Nonetheless, there's something weird about how Mac OS X's Unix shell lists files: it distinguishes between uppercase and lowercase letters when alphabetizing, and uppercase words come first in the ASCII code page. Thus, a file listing at the command line will be sorted differently from one in the Finder, with all the

items whose names begin with capital letters listed before the ones in lowercase.

NOTE: Mac OS X's case-handling behavior is a feature of the HFS+ (Mac OS Extended) filesystem, Apple's standard disk format. Other filesystem types, such as UFS and ZFS, are available for experts; because they're pure Unix filesystems, their case handling is in the Unix vein: case sensitive and case preserving.

Special Characters to Avoid in Filenames

Every operating system has some restrictions it places on what characters you can use in filenames, and Mac OS X is no exception. In fact, it actually has more complexity to worry about than most systems, if you're going to be working with the shell as much as with the Finder.

Like other Unixes, the command-line portion of Mac OS X forbids you from using the forward-slash (/) character in filenames. This is because slashes are used to delimit directory names in paths; for example, `/Users/btiemann/Documents/File1.txt` represents a file four folders down from the system root. I can't name a file `Taxes/2006.pdf`, because the system would think I'm talking about a subfolder called `Taxes` with a file called `2006.pdf` inside it.

Okay, so slashes are fairly easy to avoid. But if you're checking my work, you'll have noticed that you can create a "Taxes/2006.pdf" file without any trouble in the Finder. What gives?

The answer is that, historically, the classic (pre-OS X) Mac OS allowed slashes—because it used the colon (:) as its path delimiter, not the slash. When Mac OS X came out, rather than forcing everyone to go through an upgrade procedure to rename all their files with slashes in the names, it simply interpreted those files on the command line with colons instead of slashes. Similarly, if you create a file at the command line with a colon in it, it will show up as a slash in the Finder. Try it: type **`touch blah:foo`** at the command line, and watch the file "blah/foo" appear in the corresponding Finder window.

The upshot is that you can't use colons in the Finder, and you can't use slashes in the shell—but the reverse in both cases is perfectly legal. If you have trouble keeping this straight, don't worry: you're not alone. (Or should that be "don't worry/you're not alone"?)

That's not where the inconveniences end, unfortunately. There's also the unpleasant matter of spaces, apostrophes, quotes, dashes, asterisks, and other characters that seem perfectly natural as names of documents but that will cause you fits if you try to work with them on the command line. Each of the character classes in Table 3.1 has a special meaning for Unix, one that doesn't normally impinge on your life in the Finder, but that can make the shell fall over and twitch if you don't know what you're doing.

Table 3.1 **Avoiding Special Characters in Filenames**

Character	Meaning
Space	Separator between command arguments
/	Path delimiter
\	Escapes the following character
-	Can indicate a command option
[]	Shell scripting tokens
{}	Shell scripting tokens
*	Wildcard (multiple characters)
?	Wildcard (single character)
'	Command argument grouping delimiter
"	Command argument grouping delimiter

To use any of the preceding characters in a filename in the shell, you have to *escape* it—precede it with a backslash character, which tells the shell to treat the next character in the filename literally, not as a special command character. For instance, suppose you have a file called `My "Road Trip" CDs.txt` that you want to address using a shell command (`ls`). You'd have to write the command like this:

```
Silver:~ btiemann$ ls My\ \"Road\ Trip\"\ CDs.txt
```

This tells the shell that the spaces and quotes are part of the filename, not separate arguments for the `ls` command. Otherwise, `ls` would be trying to list three separate files: one called `My`, another called `Road Trip`, and a third called `CDs.txt`.

TIP: The command-line completion feature of the `bash` shell can mitigate most of the pain associated with special characters in filenames. For example, type **`ls`**

`My` and then press Tab, and unless other files in the folder have names that start with `My`, `bash` will automatically fill out the rest of the file with all the special characters escaped for you. This helps only when you're addressing existing files, though; you still have to do all the escaping yourself if you're creating a new file or applying a new name.

To keep your command-line life simple, I'd recommend that you just avoid using weird characters like the ones described here. For files that you plan on using only in the GUI side of Mac OS X, it's okay to use whatever letters the Finder will accept. But your life on the command line will be a lot happier if you leave out the spaces, quotes, and asterisks in the files you create there.

Wildcards and What They Mean

Wait. What? Wildcards? What's that about?

Unless your computing career has encompassed Unix/Linux or MS-DOS, wildcards will be something new to you. They're unique to command-line operating system environments and are also a key part of their usability. Wildcards are what allow you to specify groups of files all at once, based on similarities in their filenames.

The asterisk (*) character can be used to represent any contiguous series of characters, and the question mark (?) can represent any single character. Using these wildcard characters, you can perform repetitive or tedious tasks on large groups of files all at once, instead

of having to do it over and over, once per file. For instance, consider the following list of files:

```
Picture01.jpg
Picture02.jpg
Picture03.jpg
Pics.txt
```

Suppose you wanted to get a list of only the JPEG files in this directory. That could be accomplished in any of several ways:

```
Silver:~ btiemann$ ls *.jpg
Silver:~ btiemann$ ls Picture0?.jpg
Silver:~ btiemann$ ls Pict*
```

As you can see, wildcards in Unix don't discriminate between the filename and the extension; the asterisk wildcard covers the `.jpg` part of the affected files as well as the unspecified portion before the period. This differs from the MS-DOS way, in which you had to specify `*.*` to refer to all the files in a directory. In Mac OS X and other Unixes, you can use `*` to cover everything.

Another kind of wildcard that gives you more precise control is the brackets (`[]`), which lets you specify a set of matching characters (instead of the "any character" that the ? wildcard implies). Any characters specified within the brackets are potential matches. For example:

```
Silver:~ btiemann$ ls Picture0[13].jpg
Picture01.jpg          Picture03.jpg
```

NOTE: Wildcards don't work on "hidden" files, which you'll learn more about in Chapter 4, "Basic Unix Commands." In other words, a "hidden" file (a file

whose name begins with a period, such as `.login`) will not appear in a listing generated by `ls *`, nor will it be deleted by `rm *`. You have to delete hidden files manually, one by one.

You might be accustomed to selecting large groups of files in a Finder window, visually, to move or delete them all in one fell swoop. This might seem a more direct solution than wildcards, and in many cases it is. But if the files are all named similarly enough that they can all be described using a wildcard or two, and if the Finder can't group them efficiently, you might find that using the Terminal and wildcards can save you some time over doing it the Finder's way.

Conclusion

If this book is your first introduction to Unix, you'll be tackling it with less first-hand guidance than I had when I was first shown a SunOS login shell in 1994. It's a rare and adventurous soul who dives straight into the world of the Unix shell and tries to learn all about it on his own, without a mentor or guru handy to point out the pitfalls and offer helpful shortcuts. There are so many of these potential traps that even a thick book dedicated to Unix can't cover them all; only experience can give you the familiarity you need to be completely fluent and efficient at the shell. Still, this chapter attempts to provide the cornerstones to an understanding of what kinds of expectations Unix has of you, the user; and in the process, you will have learned how to extrapolate from what you know to find out how to overcome the rest of the obstacles you'll encounter.