Command Line Primer

Historically, the command line interface provided a way to manipulate a computer over simple, text-based connections. In the modern era, in spite of the ability to transmit graphical user interfaces over the Internet, the command line remains a powerful tool for performing certain types of tasks.

As described previously in Before You Begin, most users interact with a command-line environment using the Terminal application, though you may also use a remote connection method such as secure shell (SSH). Each Terminal window or SSH connection provides access to the input and output of a shell process. A shell is a special command-line tool that is designed specifically to provide text-based interactive control over other *command-line tools*.

In addition to running individual tools, most shells provide some means of combining multiple tools into structured programs, called shell scripts (the subject of this book).

Different shells feature slightly different capabilities and scripting syntax. Although you can use any shell of your choice, the examples in this book assume that you are using the standard OS X shell. The standard shell is `bash` if you are running OS X v10.3 or later and `tcsh` if you are running an earlier version of the operating system.

The following sections provide some basic information and tips about using the command-line interface more effectively; they are not intended as an exhaustive reference for using the shell environments.

Note: This appendix was originally part of *Mac Technology Overview*.

Basic Shell Concepts

Before you start working in any shell environment, there are some basic features of shell scripting that you should understand. Some of these features are specific to OS X, but most are common to all platforms that support shell scripting.

Running Your First Command-Line Tool

In general, you run command-line tools that OS X provides by typing the name of the tool. (The syntax for running tools that you've added is described later in this appendix.)

For example, if you run the `ls` command, by default, it lists the files in your home directory. To run this command, type `ls` and press Return.

Most tools also can take a number of flags (sometimes called switches). For example, you can get a "long" file listing (with additional information about every file) by typing `ls -l` and pressing Return. The `-l` *flag* tells the `ls` command to change its default behavior.

Similarly, most tools take arguments. For example, to show a long listing of the files on your OS X

desktop, type `ls -l Desktop` and press Return. In that command, the word `Desktop` is an argument that is the name of the folder that contains the contents of your OS X desktop.

In addition, some tools have flags that take flag-specific arguments in addition to the main arguments to the tool as a whole.

Specifying Files and Directories

Most commands in the shell operate on files and directories, the locations of which are identified by paths. The directory names that make up a path are separated by forward-slash characters. For example, the Terminal program is in the `Utilities` folder within the `Applications` folder at the top level of your hard drive. Its path is `/Applications/Utilities/Terminal.app`.

The shell (along with, for that matter, all other UNIX applications and tools) also has a notion of a current working directory. When you specify a filename or path that does not start with a slash, that path is assumed to be relative to this directory. For example, if you type `cat foo`, the `cat` command prints the contents of the file `foo` in the current directory. You can change the current directory using the `cd` command.

Finally, the shell supports a number of directory names that have a special meaning.

Table A-1 lists some of the standard shortcuts used to represent specific directories in the system. Because they are based on context, these shortcuts eliminate the need to type full paths in many situations.

Table A-1 Special path characters and their meaning

Path string	Description
`.`	The `.` directory (single period) is a special directory that, when accessed, points to the current working directory. This value is often used as a shortcut to eliminate the need to type in a full path when running a command. For example, if you type `./mytool` and press return, you are running the `mytool` command in the current directory (if such a tool exists).
`..`	The `..` directory (two periods) is a special directory that, when accessed, points to the directory that contains the current directory (called its *parent directory*). This directory is used for navigating up one level towards the top of the directory hierarchy. For example, the path `../Test` is a file or directory (named `Test`) that is a sibling of the current directory. **Note:** Depending on the shell, if you follow a symbolic link into a subdirectory, typing `cd ..` directory will either take you back to the directory you came from or will take you to the parent of the current directory.
`~` or	At the beginning of a path, the tilde character represents the home directory of the specified user, or the currently logged in user if no user is specified. (Unlike `.` and `..`, this is not an actual directory, but a substitution performed by the shell.) For example, you can refer to the current user's `Documents` folder as `~/Documents`. Similarly, if you have another user whose short name is `frankiej`, you could access that user's Documents folder as `~frankiej/Documents` (if that user has set permissions on

`$HOME`	his or her Documents directory to allow you to see its contents). The `$HOME` environment variable can also be used to represent the current user's home directory. In OS X, the user's home directory usually resides in the `/Users` directory or on a network server.

File and directory names traditionally include only letters, numbers, hyphens, the underscore character (_), and often a period (`.`) followed by a file extension that indicates the type of file (`.txt`, for example). Most other characters, including space characters, should be avoided because they have special meaning to the shell.

Although some OS X file systems permit the use of these other characters, including spaces, you must do one of the following:

- "Escape" the character—put a backslash character (`\`) immediately before the character in the path.
- Add single or double quotation marks around the path or the portion that contains the offending characters.

For example, the path name `My Disk` can be written as `"My Disk"`, `'My Disk'`, or `My\ Disk`.

Single quotes are safer than double quotes because the shell does not do any interpretation of the contents of a single-quoted string. However, double quotes are less likely to appear in a filename, making them slightly easier to use. When in doubt, use a backslash before the character in question, or two backslashes to represent a literal backslash.

For more detailed information, see Quoting Special Characters in Flow Control, Expansion, and Parsing.

Accessing Files on Additional Volumes

On a typical UNIX system, the storage provided by local disk drives is presented as a single tree of files descending from a single root directory. This differs from the way the Finder presents local disk drives, which is as one or more volumes, with each volume acting as the root of its own directory hierarchy. To satisfy both worlds, OS X includes a hidden directory, `Volumes`, at the root of the local file system. This directory contains all of the volumes attached to the local computer.

To access the contents of other local (and many network) volumes, you prefix the volume-relative path with `/Volumes/` followed by the volume name. For example, to access the `Applications` directory on a volume named `MacOSX`, you would use the path `/Volumes/MacOSX/Applications`.

Note: To access files on the boot volume, you are not required to add volume information, since the root directory of the boot volume is `/`. Including the volume information still works, though, so if you are interacting with the shell from an application that is volume-aware, you may want to add it, if only to be consistent with the way you access other volumes. You must include the volume information for all volumes other than the boot volume.

Input And Output

Most tools take text input from the user and print text out to the user's screen. They do so using three standard file descriptors, which are created by the shell and are inherited by the program automatically. These standard file descriptors are listed in Table A-2.

Table A-2 Input and output sources for programs

File descriptor	Description
`stdin`	The standard input file descriptor is the means through which a program obtains input from the user or other tools. By default, this descriptor provides the user's keystrokes. You can also redirect the output from files or other commands to `stdin`, allowing you to control one tool with another tool.
`stdout`	The standard output file descriptor is where most tools send their output data. By default, standard output sends data back to the user. You can also redirect this output to the input of other tools.
`stderr`	The standard error file descriptor is where the program sends error messages, debug messages, and any other information that should not be considered part of the program's actual output data. By default, errors are displayed on the command line like standard output. The purpose for having a separate error descriptor is so that the user can redirect the actual output data from the tool to another tool without that data getting corrupted by non-fatal errors and warnings.

To learn more about working with these descriptors, including redirecting the output of one tool to the input of another, read Shell Input and Output.

Terminating Programs

To terminate the currently running program from the command line, press Control-C. This keyboard shortcut sends an abort (`ABRT`) signal to the currently running process. In most cases this causes the process to terminate, although some tools may install signal handlers to trap this signal and respond differently. (See Trapping Signals in Advanced Techniques for details.)

In addition, you can terminate most scripts and command-line tools by closing a Terminal window or SSH connection. This sends a hangup (`HUP`) signal to the shell, which it then passes on to the currently running program. If you want a program to continue running after you log out, you should run it using the `nohup` command, which catches that signal and does not pass it on to whatever command it invokes.

Frequently Used Commands

Shell scripting involves a mixture of built-in shell commands and standard programs that run in all shells. Although most shells offer the same basic set of commands, there are often variations in the syntax and behavior of those commands. In addition to the shell commands, OS X also provides a set of standard programs that run in all shells.

Table A-3 lists some commands that are commonly used interactively in the shell. Most of the items in this table are not specific to any given shell. For syntax and usage information for each command, see the corresponding man page. For a more in-depth list of commands and their accompanying documentation, see *OS X Man Pages*.

Table A-3 Frequently used commands and programs

Command	Meaning	Description
`cat`	(con)catenate	Prints the contents of the specified files to `stdout`.
`cd`	change directory	Changes the current working directory to the specified path.
`cp`	copy	Copies files (and directories, when using the -r option) from one location to another.
`date`	date	Displays the current date and time using the standard format. You can display this information in other formats by invoking the command with specific flags.
`echo`	echo to output	Writes its arguments to `stdout`. This command is most often used in shell scripts to print status information to the user.
`less` and `more`	pager commands	Used to scroll through the contents of a file or the results of another shell command. This command allows forward and backward navigation through the text. The `more` command got its name from the prompt “Press a key to show more....” commonly used at the end of a screenful of information. The `less` command gets its name from the idiom “less is more”.
`ls`	List	Displays the contents of the specified directory (or the current directory if no path is specified). Pass the `-a` flag to list all directory contents (including hidden files and directories). Pass the `-l` flag to display detailed information for each entry. Pass `-@` with `-l` to show extended attributes.
`mkdir`	Make Directory	Creates a new directory.

`mv`	Move	Moves files and directories from one place to another. You also use this command to rename files and directories.
`open`	Open an application or file.	You can use this command to launch applications from Terminal and optionally open files in that application.
`pwd`	Print Working Directory	Displays the full path of the current directory.
`rm`	Remove	Deletes the specified file or files. You can use pattern matching characters (such as the asterisk) to match more than one file. You can also remove directories with this command, although use of `rmdir` is preferred.
`rmdir`	Remove Directory	Deletes a directory. The directory must be empty before you delete it.
Ctrl-C	Abort	Sends an abort signal to the current command. In most cases this causes the command to terminate, although commands may install signal handlers to trap this command and respond differently.
Ctrl-Z	Suspend	Sends the SIGTSTP signal to the current command. In most cases this causes the command to be suspended, although commands may install signal handlers to trap this command and respond differently. Once suspended, you can use the `fg` builtin to bring the process back to the foreground or the `bg` builtin to continue running it in the background.
Ctrl-\	Quit	Sends the SIGQUIT signal to the current command. In most cases this causes the command to terminate, although commands may install signal handlers to trap this command and respond differently.

Environment Variables

Some programs require the use of environment variables for their execution. Environment variables are variables inherited by all programs executed in the shell's context. The shell itself uses environment variables to store information such as the name of the current user, the name of the host computer, and the paths to any executable programs. You can also create environment variables and use them to control the behavior of your program without modifying the program itself. For example, you might use an environment variable to tell your program to print debug information to the console.

To set the value of an environment variable, you use the appropriate shell command to associate a

variable name with a value. For example, to set the environment variable `MYFUNCTION` to the value `MyGetData` in the global shell environment you would type the following command in a Terminal window:

```
# In Bourne shell variants
export MYFUNCTION="MyGetData"


# In C shell variants
setenv MYFUNCTION "MyGetData"
```

When you launch an application from a shell, the application inherits much of its parent shell's environment, including any exported environment variables. This form of inheritance can be a useful way to configure the application dynamically. For example, your application can check for the presence (or value) of an environment variable and change its behavior accordingly. Different shells support different semantics for exporting environment variables, so see the man page for your preferred shell for further information.

Child processes of a shell inherit a copy of the environment of that shell. Shells do not share their environments with one another. Thus, variables you set in one Terminal window are not set in other Terminal windows. Once you close a Terminal window, any variables you set in that window are gone.

If you want the value of a variable to persist between sessions and in all Terminal windows, you must either add it to a login script or add it to your environment property list. See Before You Begin for details.

Similarly, environment variables set by tools or subshells are lost when those tools or subshells exit.

Running User-Added Commands

As mentioned previously, you can run most tools by typing their name. This is because those tools are located in specific directories that the shell searches when you type the name of a command. The shell uses the `PATH` environment variable to control where it searches for these tools. It contains a colon-delimited list of paths to search—`/usr/bin:/bin:/usr/sbin:/sbin`, for example.

If a tool is in any other directory, you must provide a path for the program to tell it where to find that tool. (For security reasons, when writing scripts, you should always specify a complete, absolute path.)

For security reasons, the current working directory is *not* part of the default search path (`PATH`), and should not be added to it. If it were, then another user on a multi-user system could trick you into running a command by adding a malicious tool with the same name as one you would typically run (such as the `ls` command) or a common misspelling thereof.

For this reason, if you need to run a tool in the current working directory, you must explicitly specify its path, either as an absolute path (starting from `/`) or as a relative path starting with a directory name (which can be the `.` directory). For example, to run the `MyCommandLineProgram` tool in the current directory, you could type `./MyCommandLineProgram` and press Return.

With the aforementioned security caveats in mind, you can add new parts (temporarily) to the value of the `PATH` environment variable by doing the following:

```
echo "$PATH"


# In Bourne shell variants
export PATH="$PATH:/my/new/path/part"


# In C shell variants
setenv PATH "$PATH:/my/new/path/part"
```

If you want the additional path components to persist between sessions and in all Terminal windows, you must either add it to a login script or add it to your environment property list. See Before You Begin for details.

Running Applications

To launch an application, you can generally either:

- **Use the `open` command.**

  ```
  open /path/to/MyApp.app
  ```

- **Run the application binary itself.**

  Type the pathname of the executable file inside the package.

  ```
  /path/to/MyApp.app/Contents/MacOS/MyApp
  ```

Note: As a general rule, if you launch a GUI application from a script, you should run that script only within Terminal or another GUI application. You cannot necessarily launch an GUI application when logged in remotely (using SSH, for example). In general, doing so is possible only if you are also logged in using the OS X GUI, and in some versions of OS X, it is disallowed entirely.

Learning About Other Commands

At the command-line level, most documentation comes in the form of man pages (short for manual). Man pages provide reference information for many shell commands, programs, and POSIX-level concepts. The manual page `manpages` describes the organization of manual, and the format and syntax of individual man pages.

To access a man page, type the `man` command followed by the name of the thing you want to look up. For example, to look up information about the `bash` shell, you would type `man bash`. The man pages are also included in the OS X Developer Library (*OS X Man Pages*).

You can also search the manual pages by keyword using the `apropos` command.

Note: Not all commands and programs have man pages. For a list of available man pages, look in the `/usr/share/man` directory or see *OS X Man Pages* in the OS X Developer Library.

Most shells have a command or man page that displays the list of commands that are built into the shell (builtins). Table A-4 lists the available shells in OS X along with the ways you can access the list of builtins for the shell.

Table A-4 Getting a list of shell builtins

Shell	Command
`bash`	`help` or `bash -c help`
`sh`	`man sh`
`csh`	`builtins`
`tcsh`	`builtins`
`zsh`	`man zshbuiltins`