



# Speaking UNIX, Part 10: Customize your shell

**Persist your preferences to recreate your shell environment**

Level: Intermediate

[Martin Streicher \(martin.streicher@gmail.com\)](mailto:martin.streicher@gmail.com), Chief Technology Officer, McClatchy Interactive

29 May 2007

You can customize the UNIX® shell to save time, to save typing, and to adapt to your style of work. *Shell startup files* capture your preferences and recreate your shell environment session after session, even machine to machine.

If you work with a tool long enough, you master its purpose. Moreover, the tool becomes an extension of yourself. Think of Gustav Klimt's brush, Louis Armstrong's trumpet, and Mark Twain's turn of phrase. If you're a virtuoso, your "tools of the trade" effortlessly channel your intent, spirit, and expression to your medium.

By now, I hope your skills have reached those of a UNIX® acolyte. You practice your command-line *katas*. You consult the omniscient oracle of *man* when you crave knowledge. And you craft command combinations that perform sheer alchemy on data. You're at ease at the command line, and the shell feels comfortable and familiar.

The next stage in your apprenticeship, Grasshopper, is to make the shell your own.

## The great and mighty shell

You've already seen many techniques to customize your shell environment:

- You can choose the UNIX shell you'd like to use. The Bourne shell is a stalwart; others, such as the Z shell, offer novelties and conveniences that you may find helpful.

To find the shells available on your UNIX system, use the command `cat /etc/shells`. To change your shell to any of the shells listed, use the `chsh` command. Here's an example to change to `/bin/zsh`, the Z shell. (Type the text shown in bold.)

```
$ cat /etc/shells
/bin/bash
/bin/csh
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
$ chsh -s /bin/zsh
```

- You can create short *aliases* to stand in for lengthy commands.
- *Environment variables*, such as `PATH` (which controls where to search for programs) and `TZ` (which specifies your time zone), persist your preferences and affect all the processes you launch.

`PATH` is especially useful. For example, if you want or need to run a local, enhanced version of Perl, you can alter your `PATH` to prefer `/usr/local/bin/perl` instead of the (typical) standard version found in `/usr/bin/perl`.

UNIX applications often use environment variables for *customization*, too. For instance, if your terminal (or emulator) is capable, you can colorize the output of `ls` (list directory contents) with the environment variables `CLICOLOR` and `LSCOLORS`.

- You can retain and recall command lines through the shell's built-in *command history*. Command histories conserve typing, allowing you to re-run an earlier command. Many shells also allow on-the-fly modification of a previous command to create a new command. For example, the Bash shell uses the caret

(`^`) character to perform substitutions:

```
$ ls -l heroes.txt
-rw-r--r--  1 strike  strike  174 Mar  1 11:25 heroes.txt
$ ^heroes^villains
ls -l villains.txt
villians.txt
```

Here, the quirky command line `^heroes^villains` substitutes the word *villains* for *heroes* in the immediately previous command (the default, if a numbered command in the history list isn't provided) and runs the result, `ls -l villains.txt`. Consult your shell's documentation for its syntax for command-line substitutions.

- You can write *shell scripts* to (re-)perform complex operations if the existing UNIX utilities and your shell's built-in features lack a feature you'd like to use regularly.

As you'll see in an upcoming "Speaking UNIX" article, you can also download and build an enormous number of additional UNIX utilities, typically provided as open source. In fact, with Google or Yahoo! and a few minutes of time, you can usually and readily find and download a suitable solution rather than create your own. (Be lazy! Spend your bonus free time watching clouds.)

Of course, with so many options for fine-tuning your shell, it would be nice if you could persist your preferences and re-use those settings time and again, from shell to shell (say, in different X terminal windows), session to session (when you log out and return to log in again), and even across multiple machines (assuming that you use the same shell on multiple platforms).

*Shell startup scripts* provide this endurance. When a shell starts and as it terminates, the shell executes a series of scripts to initialize and reset your environment, respectively. Some startup scripts are system-wide (your systems administrator configures them), and others are yours to customize freely.

Startup scripts aren't like Microsoft® Windows® INI files. As the name implies, startup scripts are true shell scripts—those little programs you write to achieve some work. In this case, the shell scripts run whenever the shell starts or terminates and affect the shell environment.

## Start me up!

Typically, each shell provides for several shell startup scripts, and each shell dictates the order in which the scripts run. At a minimum, you can expect a system-wide startup file and a personal (per-user) startup file. Think of the entire shell startup sequence as a kind of cascade: The effects of running (potentially) multiple scripts are cumulative, and you can negate or alter parameters set early in the sequence in a subsequent script.

For example, your systems administrator might set a helpful default shell prompt for the entire system—something that includes your user name, current working directory, and command history number, for instance—in the system-wide shell startup file. However, you can override this file by resetting the shell prompt to your liking in your own startup script. Otherwise, if you don't alter a system-wide setting, it persists in your shell and environment.

Typically, the earliest startup scripts are system-wide, such as `/etc/profile`, and your systems administrator manages them. System-wide startup files aren't intended as an intrusion, but rather facilitate the use of resources specific to that system. For example, if your system administrator prefers that you use a newer version of the Secure Shell (SSH) utility because it addresses a known security flaw, he or she might set each user's initial `PATH` variable to `/usr/local/bin:/bin:/usr/bin`, which prioritizes executables found in `/usr/local/bin`. (If the command isn't found in `/usr/local/bin`, the shell continues its scan in `/usr/bin`.) System-wide startup files are also used to name printers, display bulletins about planned downtime, and provide new users with reasonable shell defaults. (Don't haze the newbies.)

After the system-wide script (or scripts) runs, the shell runs user-specific startup scripts. The per-user files are the appropriate places to keep your favorite aliases, environment settings, and other preferences.

## Planning for the big Bash

The number and names of the shell startup scripts vary from one shell to another. Let's look at the startup sequence of the Bash shell, `/bin/bash`. The Bash shell is found on all flavors of UNIX and Linux®, and it is typically the default shell of new systems and users. It's also representative of many other shells and thus serves as a good demonstration. (If you use another shell, consult its documentation or man page for the names and processing order of its startup scripts.)

Bash searches for *six* startup scripts, but each of those scripts is optional. Even if all six scripts exist and are readable, Bash executes only a subset of the six in any situation.

Bash first executes `/etc/profile`, the system-wide startup file, if that file exists and the user can read it. After reading that file, Bash looks for `~/.bash_profile`, `~/.bash_login`, `~/.profile`, and `~/.bashrc`—in that order—where `~` is the shell's abbreviation for the user's home directory (also available as `$HOME`). If you exit Bash, the shell searches for `~/.bash_logout`.

Which of the six files executes depends on the "mode" of the new shell. A shell can be a *login shell*, and it might or might not be *interactive*. (A login shell is also an interactive shell; however, you can force a non-interactive shell to behave like a login shell. More on that later.)

In UNIX days of yore (a scant two decades ago), you typically accessed a UNIX machine through a dumb terminal. You would type your user ID and password at the login prompt, and the system would spawn a new login shell for your session. In this environment, a login shell was differentiated from other shell instances (such as those running a shell script) by name: The process name of each login shell was prefixed with a hyphen, as in `-bash`. This special name—a longtime UNIX artifact—tells the shell to run any special configuration for login.

An interactive shell is easier to explain: A shell is interactive if it responds to your input (standard input) and displays output (to standard out). Today, the X terminal has replaced the dumb terminal, but the convention and paradigm of shell modes remain. Usually, X terminal spawns Bash as `-bash`, which forces Bash to perform the login startup sequence.

In the case of Bash, an interactive login shell runs `/etc/profile`, if it exists. (A non-interactive shell also runs `/etc/profile` if Bash is invoked as `bash --login`.) Next, the interactive login shell looks for `~/.bash_profile` and executes this script if it exists and is readable. Otherwise, the shell continues, trying to execute `~/.bash_login`. If the latter file doesn't exist or is unreadable, Bash finally attempts to execute `~/.profile`. Bash runs only one personal startup file—the startup sequence stops immediately afterward. When a Bash login shell exits, it executes `~/.bash_logout`.

If the Bash shell is interactive but not a login shell, Bash attempts to read `~/.bashrc`. No other files are executed. If the Bash shell is non-interactive, it expands the value of the `BASH_ENV` environment variable and executes the file named.

Of course, you can provide additional settings by calling your own scripts from within Bash's standard scripts. The special shell abbreviation `.` (or its synonym `SOURCE`) executes another shell script. For example, if you want to share the settings in `~/.bashrc` between interactive login shells and interactive non-login shells, place the command:

```
. ~/.bashrc
```

in `~/.bash_profile`. When the shell encounters the dot command, it immediately executes the named shell script.

## Peering into the shell

The best way to explore the startup sequence is to create some simple shell startup files. For example, if you run the `ssh farfaraway ls` command, is the remote shell that SSH spawns on the remote system named *farfaraway* a login shell? An interactive shell? Let's find out.

Listings 1, 2, 3, and 4 show sample `/etc/profile`, `~/.bash_profile`, `~/.bashrc`, and `~/.bash_logout` files, respectively. (If these files already exist, make backups before you continue with this exercise. You need superuser privileges on your machine to change `/etc/profile`.) Use your favorite text editor to create the files as shown.

Listing 1 shows a sample `/etc/profile` script. This file is the first startup file to run (if it exists and is readable).

### Listing 1. Sample `/etc/profile` file

```
echo "Executing /etc/profile."
PATH="/bin:/sbin:/usr/bin:/usr/sbin"
export PATH
```

Listing 1 echoes a message as the script begins and sets a minimal `PATH` variable. Again, this file runs if the shell is an interactive login shell. For example, launch a new X terminal. You should see something like this:

```
Last login: Tue Apr 17 21:06:23 on ttyt1
Executing /etc/profile
(Interactive, login shell)
Executing /Users/strike/.bash_profile
(Interactive, login shell)
Including /Users/strike/.aliases
strike @ blackcat 1 $
```

Good! That's the predicted sequence when launching a new login shell in an X terminal. Notice the shell prompt: It reflects the user name, the short hostname (everything before the first dot), and the command number.

If you type `logout` or `exit` at the prompt, you should see this:

```
strike @ blackcat 31 $ logout
Executing /Users/strike/.bash_logout
(Interactive, login shell)
```

As described, the interactive login shell runs `~/.bash_logout`.

Listing 2 shows a sample `~/.bash_profile` file. This file is one option for customizing your shell at startup.

### Listing 2: Sample `~/.bash_profile` file

```
echo "Executing $HOME/.bash_profile"
echo '(Interactive, login shell)'

PS1='\u @ \h \# \$ '
export PS1

PAGER=/usr/bin/less
export PAGER

. .aliases
```

Next, let's see what happens when you launch a new shell from the prompt. The new shell is interactive, but it's not a login shell. According to the rules, `~/.bashrc` is the only file expected to run.

```
strike @ blackcat 1 $ bash
Executing /Users/strike/.bashrc
(Interactive shell)
blackcat:~ strike$
```

And, in fact, `~/.bashrc` is the only file to execute. The proof is in the prompt—the prompt at bottom is the default Bash prompt, not the one defined in `~/.bash_profile`.

To test the logout script, type `exit` (you cannot type `logout` in a non-login shell). You should see:

```
blackcat:~ strike$ exit
exit
Executing $HOME/.bash_logout
(Interactive, login shell)
strike @ blackcat 2 $
```

As an interactive login shell terminates, it executes `~/.bash_logout`. You might use this feature to remove temporary files, copy files as a simple method of backup, or perhaps even launch `rsync` to distribute any changes made in this most current session.

[Listing 3](#) shows a sample `~/.bashrc` file. This file is the initialization file for non-interactive Bash shell instances.

### Listing 3: Sample `~/.bashrc` file

```
echo "Executing $HOME/.bashrc"
echo "(Interactive shell)"

PATH="/usr/local/bin:$PATH"
export PATH
```

Here's another experiment: What kind of shell do you get when you run SSH? Let's try two variations. (You can simply use SSH to get back to your local machine—it works the same as if you were running SSH from a remote machine.) First, use SSH to log in to the remote machine:

```
strike @ blackcat 1 $ ssh blackcat
Last login: Tue Apr 17 21:17:35 2007
Executing /etc/profile
(Interactive, login shell)
Executing /Users/strike/.bash_profile
(Interactive, login shell)
Including /Users/strike/.aliases
strike @ blackcat 1 $
```

As you might expect, running SSH to access a remote machine launches a new login shell. Next, what happens when you run a command on the remote machine? Here's the answer:

```
strike @ blackcat 3 $ ssh blackcat ls
Executing /Users/strike/.bashrc
(Interactive shell)
villians.txt
heroes.txt
```

Running a command remotely using SSH spawns a non-login interactive shell. Why is it interactive? Because the standard input and the standard output of the remote command are tied to your keyboard and display, albeit through the magic of SSH.

[Listing 4](#) shows `~/.bash_logout`. This file runs as the shell terminates.

### Listing 4: Sample `~/.bash_logout` file

```
echo "Executing $HOME/.bash_logout"
echo "(Interactive, login shell)"
```

## Helpful tips for startup files

The more you use the shell, the more you can benefit from persisting your preferences in startup files. Here are some helpful tips and suggestions for organizing your Bash settings. (You can apply similar strategies to other

shells.)

- If you have settings (for example, PATH) that you want to use in every shell (regardless of its mode), place those settings in ~/.bashrc and use **source** to access the file from ~/.bash\_profile.
- If you have accounts on multiple machines (and your home directory isn't shared among them through the Network File System [NFS]), use rsync to keep your shell startup files in sync across all machines on the network.
- If you apply certain preferences depending on the host you're using—say, a different PATH if one system has special resources—place those settings in a separate file and use **source** to access it during shell startup. If you choose to use rsync to manage your files, omit the host-specific file from the file distribution list.

Of course, you can also create a global script and use conditionals and the environment variable HOSTNAME to choose the appropriate settings. (HOSTNAME is set automatically by the shell and captures the fully qualified host name.) For example, here's a useful snippet commonly found in startup files:

```
case $HOSTNAME in
  lab.area51.org)
    PATH=/opt/rocketscience/bin:$PATH
    PS1='\u @ \h \# \$ '
    export $PS1;;

  alien.area51.org)
    PATH=/opt/alien/sw/bin:$PATH;;

  saucer*)
    PATH=/opt/saucer/bin:$PATH
    PAGER=less
    export $PAGER;;

  *)
    PATH=/usr/local/bin:$PATH
esac

export $PATH
```

The construct here is a switch statement to compare the value of \$HOSTNAME against four possible values: lab.area51.org, alien.area51.org, a pattern that matches any hostnames that begin with the literal string **saucer\*** (a hostname such as *saucer-mars* would match; a hostname such as *sauce.tomato.org* would not), and everything else. Here, in the case of Bash, the asterisk (\*) is interpreted as a shell operator, not as a regular expression operator. When a match is made against one of the patterns, the statements associated with that pattern execute. Unlike other switch statements, Bash's *case* runs one set of statements only.

Finally, look at the shell startup files of other users for inspiration and to save perspiration. (Some users protect these files and their home directory, though, which precludes you from browsing.) Does Joe have a cool, useful prompt? Ask how to implement the same thing. Does Jeanette have extensive keyboard accelerators or a great collection of environment variables to eke out special features from utilities? Chat with her about her about recipes. The best source of ideas and code comes from experienced practitioners of the command line.

## Customizing your shell

Tweakers and modders, unite! You can customize your shell extensively, and after you find a setting or series of settings you like, save them in a startup file to re-use again and again. Use rsync or a similar tool to propagate your environment from one machine to another.

Your lesson is done. Time for more katas.

## Resources

### Learn

- [Speaking UNIX](#): Check out other parts in this series.
- Check out other articles and tutorials written by Martin Streicher:
  - [Across developerWorks and IBM](#)
- Search the AIX and UNIX library by topic:
  - [System administration](#)
  - [Application development](#)
  - [Performance](#)
  - [Porting](#)
  - [Security](#)
  - [Tips](#)
  - [Tools and utilities](#)
  - [Java™ technology](#)
  - [Linux](#)
  - [Open source](#)
- [AIX and UNIX](#): The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- [New to AIX and UNIX?](#): Visit the "New to AIX and UNIX" page to learn more about AIX and UNIX.
- [AIX 5L™ Wiki](#): Discover a collaborative environment for technical information related to AIX.
- [Safari bookstore](#): Visit this e-reference library to find specific technical resources.
- [developerWorks technical events and webcasts](#): Stay current with developerWorks technical events and webcasts.
- [Podcasts](#): Tune in and catch up with IBM technical experts.

## Get products and technologies

- [IBM trial software](#): Build your next development project with software for download directly from developerWorks.

## Discuss

- Participate in the [developerWorks blogs](#) and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
  - [AIX 5L—technical forum](#)
  - [AIX for Developers Forum](#)
  - [Cluster Systems Management](#)
  - [IBM Support Assistant](#)
  - [Performance Tools—technical](#)
  - [Virtualization—technical](#)
  - [More AIX and UNIX forums](#)

## About the author



Martin Streicher is the Chief Technology Officer of McClatchy Interactive and the Editor-in-Chief of *Linux Magazine*. Martin holds a Masters of Science degree in computer science from Purdue University and has been programming UNIX-like systems since 1986. You can reach Martin at [martin.streicher@gmail.com](mailto:martin.streicher@gmail.com).

## Share this....



[Digg this story](#)



[del.icio.us](#)



[Slashdot it!](#)

IBM, AIX, and AIX 5L are registered trademarks of International Business Machines Corporation in the United States, other countries, or both. AIX is a registered trademark of IBM in the United States. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries. Other company, product, or service names may be trademarks or service marks of others.