

Scripting

by Chris Herborth

In Chapter 6 we introduced you to the GNU `bash` shell that you'll find in BeOS's Terminal window. With any luck, you've been spending a few minutes here and there playing with the shell, experimenting and getting an idea of the things you can do from a command line, (or maybe you were already familiar with using a shell).

If you were frightened by Chapter 6, you might want to postpone reading this chapter until you've gotten familiar with the shell and are ready to make it even more useful.

What Is Scripting?

Good question! These days, there's a fine line (or no line) between “scripting” and “programming.”

Wait, don't run away! Yes, I just said the “p” word, but don't let that scare you. You don't need a degree in computer science to write useful scripts—in fact, if you've used one of the many huge, bloated software packages available for other platforms, you've probably *already* done some “programming” in the form of macros. Feel free to give yourself a pat on the back if you've just learned that you're a programmer.

People spend hours of otherwise useful time arguing about the definition of “scripting” versus “programming” languages. By the end of the argument, the combatants are down to discussing details that nobody cares about, and they *still* don't have a good definition; it usually has more to do with what you're using the language for *at the time*. However, many people say, and we're going to assume in this chapter, that a scripting language is about controlling other applications, not programming a new application. (Unless that new application is for controlling other applications...you should be starting to see how annoying this discussion gets.)

When we talk about “scripting” in BeOS, we're usually talking about two different (but sometimes related) things: shell scripts and BeOS application scripting.

Even though scripting doesn't require programming skills, learning to write scripts can be a great way to get a handle on basic concepts that you can use later on if you *do* decide to learn to program!

Shell Scripts, Application Scripts, What's the Diff?

If you're new to the whole concept of scripting, it may be a little tough at first to sort out the difference between shell scripting and application scripting, and to understand why we've organized this chapter the way we have. These two concepts are both completely separate and tightly linked. As you learned in Chapter 6, *The Terminal*, there are hundreds of commands available to the `bash` shell. As you'll learn in this chapter, you can string these commands together and save them in text files to automate anything you do from the shell. That's shell scripting—automating your operating system with command-line tools.

“Application scripting” refers to controlling your BeOS applications, both third-party applications and those that came with your system. The BeOS architecture makes this powerful functionality available for you with *any* application in your system—you don't have to rely on vendor-specific macro languages. All you need to script BeOS applications is the ability to send those applications system messages, called BMessages, and the ability to find out which messages a given application will respond to.

As you'll learn in this chapter, the ability to send system messages will eventually be built into most of the major scripting languages available for BeOS, which will let you use any language to create application scripts for BeOS. Until that time, though, we need to use a mini-language specifically designed to send BeOS system messages, and control that language from within scripts. That package is a command-line utility called `hey`.

In this chapter, we're going to take a brief look at the principles behind both types of scripting, and then dig more deeply into each one. We'll use the shell and simple shell scripts to demonstrate the use of `hey` and give you a taste of the power of BeOS application scripting. Shell scripting is a mighty powerful tool in its own right, and combined or extended with BeOS-specific scripting opportunities, is a veritable Swiss Army knife

capable of making your computer do what it's supposed to do: make your life easier and take care of the grunt work on your behalf.

- ? <<format as note>>When it comes to controlling a GUI application from the shell with hey, the shell is just another programming language. hey gives the shell the ability to send messages to GUI applications, and that's the *only* requirement to make any programming or scripting language into a BeOS application scripting dynamo.
- ?
- ? Let me say that again so you don't miss it: There's nothing special about the shell and hey. *Any* programming language that can send messages will be able to remotely control GUI applications, and thus will be a candidate for BeOS application scripting.

<<end note>>

Shell Scripts

A “shell script” is just a text file (usually with a filetype of `text/plain` or `text/x-script.sh`) that's been marked as an executable with the `chmod` command. (Don't worry, we'll show you how to do this just a little later.) Inside this text file will be one or more shell commands—normal commands that you could just type into the Terminal at the command line. For example, a really simple (and somewhat pointless) shell script to print “hello world” could look like this:

```
#!/bin/sh
#
# Print a friendly message:

echo hello world
```

Wait a minute, what's that crap at the beginning? Well, it's magic, and we'll talk about it in detail in the *Magic Cookies* section.

Most shell scripts are a lot more complicated than that; if they were that simple, you'd just type the commands into the shell. Shell scripts are a way of sticking a sequence of commands together in a convenient package; this lets you reuse long sequences of commands, to do almost anything.

BeOS Application Scripting

So, if a shell script is just a bunch of shell commands, what's BeOS application scripting? It's certainly got a more impressive name. To

explain it in more detail, I've got to wax poetic and discuss how some other operating systems work, how applications communicate in BeOS, and some other details.

Messages As mentioned earlier in this chapter and elsewhere in this book, BeOS does much of its work by sending small, well-defined messages (BMessages) between the various running applications and system servers. This helps distribute the workload, makes the system stable (by keeping the all-important servers separate from the user's applications), and generally makes life better for everyone running BeOS.

For example, the messages sent between each application and the system's Application Server cause the `app_server` to put windows on your screen, redraw parts of your windows (also known as "views"), and do various other useful things. In return, the `app_server` sends messages to the application every time the user presses a key, clicks the mouse, opens a menu, or closes a window.

Wouldn't it be useful if you could send commands to your applications just like the `app_server` does? In other words, imagine that you could send messages to an application telling it to close a window, open a file, make a menu selection, or do anything you would normally do if you were using the application directly —you'd be able to automate your applications and make them do repetitive, boring tasks without having to sit there and do them yourself.

That's exactly what application scripting is! Every single application running on BeOS can accept and respond to “suites” of scripting commands, whether or not the programmer added special code to handle this. It's all automatic.

In BeOS, application scripting is done by sending messages to running applications. If necessary, they can send back information in a reply message.

Scripting Wars

Every now and then, a seemingly innocent question pops up on the various BeOS email lists: “Why doesn't BeOS come with a native scripting language of its own?”

Usually this is accompanied by some pining for the poster's favorite operating system, usually OS/2 or the Amiga, both of which happen to have the REXX scripting language built in. Many applications for both OSs could be scripted using REXX, and developers could distribute REXX programs with their applications as examples of their scripting support, similar to the way MacOS applications can be controlled with AppleScript, the

way Windows applications can be controlled with VisualBASIC, and the way some UNIX applications can be controlled with Tcl.

This discussion almost instantly degenerates into an argument about the merits of various scripting languages and operating systems, and is immediately banned from the mailing list because it has nothing to do with the list's usual subject (such as using BeOS or developing BeOS applications).

Everyone is tired of these Scripting Wars; all of the list old-timers have either given up trying to convert everyone else to The One True Scripting Language, or they've come around to believing that BeOS is doing the Right Thing.

Which, of course, it is. By using the standard message-passing objects and functions already present in BeOS, you can use *any* scripting or programming language to control *any* BeOS application. The language and the application don't have to be specially designed to talk to each other, and the user can use whatever language they feel comfortable with, as long as it can send messages.

It's hard to argue against this sort of flexibility, though some people try. Usually they complain about not having a standard scripting language, which means that they have to write their sample scripts in all languages or they can't distribute any sample scripts at all because they have no way of knowing what scripting language the user is going to have installed.

But what these detractors fail to understand is that BeOS *does* come with a standard scripting language: the `bash` shell!

This doesn't mean that `bash` is the only scripting language that you can use with BeOS; again, you can use any language that lets you send messages to running applications.

Application Scripting Languages for BeOS

At the time of this writing, there were quite a few languages available on BeWare, and there will probably be a lot more available when you read this. Although not all of these languages will be able to send scripting messages to BeOS applications, popular ones like Python and Perl should be able to soon (possibly by the time you read this).

Luckily, BeOS's strong POSIX support means that it's usually fairly easy to port any of the existing scripting languages that are out there on the Net to BeOS.

Here's a quick (and hopefully unbiased!) overview of some of the languages available on BeWare:

- Guile
- Hope
- HU-Prolog
- Oberon-2
- Perl
- Python
- Ruby
- SmallEiffel
- Tcl

Most of these are pretty rare; the Big Three of scripting are Perl, Python, and Tcl. (Guile will probably become more popular as more GNU applications support it.) Each of the Big Three seems to have more regular users than all of the others put together, if you believe the polls you find now and then on the Web.

- ? GNU (short for GNU's Not UNIX) is a set of UNIX-compatible software developed by the Free Software Foundation (FSF). Anyone can download, modify, and redistribute GNU software, but they can't limit further distribution. Richard Stallman started the GNU project in 1983 at MIT.

<<prod: format as note>>

<<end note>>

Rare Languages

Guile, Hope, HU-Prolog, Oberon-2, Ruby, and SmallEiffel seem to be fairly rare and used only for special purposes or by enthusiasts. You probably won't need to know much about them; if you've got a special use for one of these languages, then hopefully you'll already know what you're doing.

Perl

<http://www.perl.org/>

Larry Wall's Perl is a very popular (and some people say challenging) language used by many systems administrators and Webmasters to ease their day-to-day work and manage Web sites. Known chiefly for its powerful text-manipulation capabilities, it excels at performing tricky search-and-replace operations over lots of text files (which probably explains why it's so popular with Webmasters wrangling with huge mountains of HTML). The CGI (Common Gateway Interface) scripts that work behind the scenes of most Web-based forms, such as search engines and questionnaires, are often written in Perl.

Genetically speaking, Perl is a hybrid of the Bourne shell and the `awk` language (both discussed in Chapter 6), with some C thrown in for good measure. Due to its heavy use of regular expressions and its tendency to give the programmer several ways to do the same thing, Perl code can be difficult to read (my favorite description is that it looks like someone sat on your keyboard). On the other hand, this complexity can make for some very powerful and very short programs.

Perl can't send BeOS messages yet, but it's only a matter of time before this popular language becomes capable of controlling GUI applications. Be sure to check Perl's entry on BeWare to see if a new version's been released!

Python

<http://www.python.org/>

Named after the British comedy troupe and not the snake of the same name, Python is an easy-to-use, object-oriented language that is well suited to application scripting, even on systems that don't support a rich messaging model like BeOS does.

Python programs look a little like pseudo-code (I can hear the computer science students groaning in the back) that actually runs. Like BeOS, Python supports most of the current industry buzzwords for programming languages (modules, classes, exceptions, very high-level dynamic data types, and dynamic typing, not to mention being interpreted and interactive). If these terms mean nothing to you, don't sweat it. Just as you don't have to know how your engine works in order to drive your car, you can create working scripts in Python (or any of these languages) without first understanding their every nuance.

Python is the first language to support BeOS application scripting. For the Fall '98 BeOS Masters Awards, I developed `heymodule`, a Python add-on that lets you write `hey`-style programs within the confines of the Python

interpreter. For more information about `heymodule` (it's got good documentation and several examples, including a Python version of the big email checking script that you'll find at the end of this chapter), look in BeWare's Languages section (<http://www.be.com/beware/Languages.html>). Python installation is covered in Chapter 15, *Other Goodies*.

Tcl

<http://www.tclconsortium.org/>

Tcl (pronounced “tickle” by some Unix weenies) stands for “tool control language,” as Tcl was originally designed to be embedded inside applications on Unix and used as a scripting language for those applications.

Tcl's distinguishing characteristic is that it treats everything as a string of text. This slows down some things (don't try doing lots of math, for example) but also makes it easy to embed Tcl commands inside a Tcl program, a file, or an application.

To quote from the Tcl FAQ's answer to “What is Tcl?,” “Tcl was designed with the philosophy that one should actually use two or more languages when designing large software systems. One for manipulating complex internal data structures, or where performance is key, and another, such as Tcl, for writing smallish scripts that tie together the other pieces, providing hooks for the user to extend.” This is exactly the sort of thing we're trying to accomplish with BeOS's application scripting, although in a language-neutral way.

It might be a while before the BeOS version of Tcl can do GUI scripting; I'm not sure if anyone is working on extending it to work with messages.

Setting Up a Script

No matter what kind of scripting you are going to do, shell or application, you have to set up your scripts in a certain way for the system to be able to find them, execute them, and run them as scripts. In this section we'll show you how to change plain text files into executable scripts, no matter what language you're using.

As we said above, you can use any language to write scripts in BeOS. For most of the examples in this chapter, however, we'll be using the

`bash` shell and the command-line tool `hey` to work with application scripting. The biggest advantage to using `bash` for your scripts is that you can easily share them with any other BeOS user—`bash` comes built into every copy of BeOS.

What makes a normal text file full of commands a shell script? Well, there are two parts: the magic cookie and the x bit.

-

The Magic Cookie

- ? The “magic cookie” in a shell script isn’t the same kind of cookie used on the Internet; it’s just an easily recognizable sequence that the system uses to indicate an executable script. It won’t follow your every move, store your credit card information, or fill your disk with mysterious files from who-knows-where.

```
<<<prod: format as a note>>>
<<<prod: end of note>>>
```

Remember our previous simple shell script example:

```
#!/bin/sh
#
# Print a friendly message:

echo hello world
```

That first line is the magic cookie. Normally, when the shell sees a “#” character, it ignores the rest of the line; this lets you put explanatory notes—or “comments”—in your shell scripts. If the second character of a comment in the first line of a shell script is “!”, it isn’t a normal comment anymore—it tells the shell that this is a script of some sort.

The rest of the magic cookie line tells the script where to find the program that will interpret the commands in the script. The sample script is a shell script, and will be run by the shell, which lives at `/bin/sh`.

You also need a magic cookie line for any other scripts that need a scripting interpreter. You would set these in the same way, substituting the appropriate script interpreter for `/bin/sh`.

For example, this sample script needs to run in the Python interpreter:

```
#!/boot/home/config/bin/python
#
# Print a friendly message:

print "hello world"
```

What If Someone Installed Python Somewhere Else?

The default location for Python is `/boot/home/config/bin`, but some people like to rearrange things the way *they* want them. If Python isn't at the default location, your script will need to find it. You can use a command named `env` (which stands for “environment”) to will search for something for you:

```
#!/bin/env python
#
# Print a friendly message:

print "hello world"
```

Now `env` will search for Python and tell it to run this script. This script will run on any system that has Python installed, as long as it's installed in one of the system's executable search paths (see Chapter 6 for more information about the `PATH` environment variable, which controls your search path).

Keeping hard-coded paths out of your scripts is a *very* good idea, especially if you intend to distribute your scripts. Nobody will ever have a system set up exactly like yours. The usual way to let a user customize a script is to read your paths from command-line arguments, environment variables, or a config file. Some of the books mentioned in Chapter 6's *Learning More* section will tell you all you need to know about doing this sort of thing.

Comments that start with `#!` anywhere in the file other than on the first line are just normal comments. Magic cookies can only appear on the first line of a shell script.

The x Bit

The x bit has nothing to do with Mulder and Scully. As you'll recall from Chapter 6, *The Terminal*, every file has some Unix-style permission bits associated with it, indicating who can read the file, write to it, and so on. One of these bits is the x bit, which indicates if file can be executed.

To set the x bit on your shell script, use the `chmod` command:

```
$ chmod +x script_name
```

Why not try it now? Open up StyledEdit and create a file named `hello_script`; type our sample shell script and save it:

```
#!/bin/sh
#
# Print a friendly message:

echo hello world
```

Now open a Terminal, find `hello_script`, and check its permissions:

```
$ ls -l hello_script
```

You should see something like this:

```
-rw-r--r--  1 chrish  techies          28 Aug 10 16:58 hello_script
```

Now set its x bit:

```
$ chmod +x hello_script
```

If you check the file's permissions again, you should see that the x bit is now set:

```
$ ls -l hello_script
-rwxr-xr-x  1 chrish  techies          28 Aug 10 16:58 hello_script
```

You can now run it just by typing its name:

```
$ hello_script
```

If nothing happens or you get an error message, either permissions haven't been set correctly, or the script is not in one of the directories in your `PATH`. If the shell prints:

```
hello world
```

then congratulations—you've just written a program for BeOS!

Other Ways of Using Scripts

Say you've written a really great shell script that does something useful (we'll start working on that soon!), and you'd like to keep it around and use

it whenever you need it. Do you always have to work in the directory where the script is located?

Nope! As we've seen, the system treats shell scripts as normal executable programs (assuming they've got the right magic cookie and x bit). The `PATH` environment variable (see Chapter 6, *The Terminal*) helps the shell find programs, and there's a standard directory for your own custom programs: `/boot/home/config/bin..` This directory is always in the `PATH`, so if you want to reuse your super wonder script, copy or move it there:

```
$ cp super_wonder_script ~/config/bin
```

Remember, in the shell you can use `~` as a shortcut for your home directory, which is set to `/boot/home`. Putting your scripts in `/boot/home/config/bin` lets you use them no matter where you are in the Terminal.

Once your scripts are in this folder, which is in the system `PATH`, you can use them from any other location in your filesystem.

- ? There's a handy program on BeWare called `xicon` that helps you run scripts from the Tracker; we'll talk about it in *Making Your Scripts Run from the Tracker*, later in this chapter.

```
<<< prod: format as Note >>>
<<< prod: end of Note >>>
```

Shell Scripting 101

Now that we've spent some time going on about the difference between shell and application scripting, why we'll use `bash` with `hey` for application scripting, and how to set up a script, isn't it about time we started looking at something more useful? Well, your wish is my command.

In Chapter 6, *The Terminal*, you learned how to copy, move, and rename files using the `cp` and `mv` commands. You probably thought, "Great, I can use this for lots of things!" and merrily started rearranging your filesystem.

Soon, though, you'll run into a problem: These commands only work on one file or directory at a time (for renaming) or one destination at a time (for copying and moving). What if you've got a big directory full of files

that you want to rename? What if some silly program (or user!) has set a whole bunch of files in a bunch of directories to the wrong filetype, and you want to fix them *now* instead of waiting around for the Registrar to do it?

To do these sorts of tricky things, you're going to have to learn a little shell programming. Don't worry, though—it's easy!

Looping

Say you've got a directory full of files you've downloaded from the Web; they're all text files, but none of them have file extensions. Since you like to share your files with other operating systems (possibly even on the same computer), you want to give them all a `.txt` extension so operating systems without a studly MIME typing scheme will have a clue what to do with the file.

Setting Up a Loop You could rename each file from the Tracker, or from a Terminal using `mv`, but that'd take ages if there are lots of files. If you'd like to save some time (and your brain), you could rename all the files in one fell swoop using a `for` loop in the Terminal:

```
$ for i in * ; do mv "$i" "$i.txt" ; done
```

or, if you're writing it as a shell script, you could add some extra white space to make it more readable:

```
#!/bin/sh
for i in * ; do
    mv "$i" "$i.txt"
done
```

<<< prod: format as Note >>>

If you do this in the shell, it'll look something like this (it won't let you add tabs to make it more readable):

```
$ for i in * ; do
> mv "$i" "$i.txt"
> done
```

Note the “secondary” prompt that you'll get when you continue a command over several lines; that's the `>` in the example.

<<< prod: end of Note >>>

<<< prod: format as a Tip >>>

Just as the `PS1` environment variable controls the default prompt (which starts life as `$`), the `PS2` environment variable controls the secondary prompt (which starts life as `>`) that you get when you continue a command across lines.

For example, if you do this:

```
export PS1="Keeps going: "  
export PS2="... and going: "
```

and type that `for` loop again, you'll see something like this:

```
Keeps going: for i in * ; do  
... and going: mv "$i" "$i.txt"  
... and going: done  
<<< prod: end Tip >>>
```

Don't panic—I'm about to explain every detail of the syntax used in this construct. This `for` loop takes every file in the current directory (represented by the `*` wildcard) and runs through the commands between `do` and `done`; every pass through the commands assigns one of the filenames to the variable named `i` (variables are explained in Chapter 6, *The Terminal*). For example, if you've got files named `chris`, `henry`, `scot`, and `simon` in a directory, the script will treat each of those files in turn. On the first trip though, the `mv` command will assign `chris` to the variable `i`, then rename `$i` to `$i.txt`. In other words, `chris` gets renamed to `chris.txt`. The second pass through will rename `henry` to `henry.txt`, and so on.

I've put quotes around the variable (`"$i"` instead of just `$i`) in case there are any files with spaces in their names. If you don't, the shell will think that there's one argument per part of the filename (a file named "this is a test" would be seen as four arguments, for example). This isn't too important in an `echo` command, but will cause the command to fail (or do something unexpected!) with `cp` or `mv`.

The `for` Loop Defined The general form of the `for` loop (you can type `help for` in the shell if you need a reminder) is

```
for NAME in WORDS ; do  
    COMMANDS  
done
```

NAME is the name of the variable (it can be any combination of letters, numbers, and the underscore character; I usually use `i` because I'm too lazy to think up a better name or type all of `index`), which will be assigned one of the WORDS on each pass through the set of COMMANDS. COMMANDS can be several shell commands. For example, if you're paranoid (like I am), or just like to get some feedback so you know your script is doing what you intend, you can expand the renaming loop to look like this:

```
for i in * ; do
    echo Renaming "$i"...
    mv "$i" "$i.txt"
done
```

When running this script from our example directory, the Terminal would report the following:

```
Renaming chris...
Renaming henry...
Renaming scot...
Renaming simon...
```

A Better Version of a for Loop If you find you're doing this sort of thing a lot, you could create a slightly better version of this script. A full shell script version of this set of commands might look like this:

```
#!/bin/sh
#
# Rename all the dropped files to have .txt extensions,
# then give them the text/plain filetype.

# Loop through all of the command-line arguments, which we
# collect using the special variable $@, described below.

for i in "$@" ; do

    # Rename the file:

    echo Renaming "$i"...
    mv "$i" "$i.txt"

    # Make sure it's got the right filetype:

    settype -t text/plain "$i.txt"
done
```

At the start, we've got the magic cookie for a shell script, plus a couple lines of comments to remind us what this script does. After that, there's one new thing in this shell script: The `$@` in the `for` loop is a variable that holds all of the arguments that we specified when we called the script

(arguments are covered in Chapter 6, *The Terminal*, in the *Basic Shell Syntax* section). Going through the `for` loop, the commands will be executed for every argument, which is exactly what we want. Inside the loop, each argument is renamed to end in `.txt`, and is then given a filetype of `text/plain`.

Save this file as `txt_renamer`, use `chmod +x txt_renamer` to make it executable, and store it in `~/config/bin`; now you can use it to rename text files (and give them the right filetype) any time you want by invoking it and passing it filenames like this:

```
$ txt_renamer chris henry scot simon
```

Substitutions

Imagine that that after we've renamed our directory full of files and set the type to plain text, we discover that they're all really HTML documents and that their new names and types are going to confuse everyone.

The best thing to do would be to rename these files to have normal `.html` extensions and set the filetype to `text/html`. But watch out— the first thing that comes to mind is

```
#!/bin/sh
for i in *.txt ; do
    mv "$i" "$i.html"
done
```

But if we go this route we'll end up with a bunch of files named `whatever.txt.html`. Not quite what we wanted. Wouldn't it be nice if we could strip off the `.txt` extension before we added `.html`? If you remember reading about `sed` in Chapter 6, you might think we could use it somehow; a command like `sed -e s/.txt/.html/` will replace `.txt` with `.html` for us. This starts to get tricky, though, and all we wanted to do was something simple.

<<prod: format as note>>

If you're coming to `bash` from the DOS world, you may be surprised that such a simple thing as renaming a batch of files should be so involved. After all, DOS lets you type `ren *.txt *.html` and be done with it (and no, simply typing `mv *.txt *.html` into the shell does not work; this will attempt to move all the `.txt` files and all of the `.html` files into the last `.html` file!). As you've no doubt realized by now, `bash` lets you do things that DOS could never even dream about. Unfortunately, there are side effects to the shell's flexibility and power, and this little file renaming

quirk is one of them. Rest assured, though, that examples like this—where `bash` actually makes things harder than they are in DOS—are few and far between, and that the almost unlimited power you get in return is well worth any extra effort. Plus, we're going to learn a lot about the shell's possibilities by working on this renamer.

<<end note>>

Simple Substitution Luckily, `bash` gives us a simple way of doing what we want:

```
#!/bin/sh
for i in *.txt ; do
    mv "$i" "${i%.txt}.html"
done
```

The tricky bit of this script is in the `mv` command; we've stuck in curly brackets with some extra stuff.

These curly brackets turn into a *substitution*; which the shell will use to transform text according to your commands. In this case, the `%` command is used to strip some text from the end of the `i` variable's contents. It's called a "substitution" because the transformed text is substituted for the original text.

The substitution looks like this:

```
${i%.txt}
```

and if the contents of the `i` variable match `.txt`, you'll get the contents of `i` without `.txt` on the end. Let's play with this in the shell a bit to see what I mean:

```
$ TEXT="hello there"
```

```
$ echo ${TEXT%what}
hello there
```

```
$ echo ${TEXT%there}
hello
```

```
$ echo ${TEXT%the}
hello there
```

You'll get back a "hello there" the first time because "what" doesn't match anything at the end of `$TEXT`. The second time, you'll get "hello" (actually "hello " with a space after it) because "there" *does* match the end of the text. The third time, you might expect to get "hello re", but you don't... "the" doesn't match the *end* of the text, so nothing happens.

The general form of the % text-stripper is:

```
${variable%text}
```

and it removes the given text from the end of variable's contents.

Combining Substitutions with a for Loop This construct is handy because it lets us strip off unwanted bits at the end of things, such as the incorrect .txt extension in our example. If we try this out with our example directory of files:

```
$ ls  
  
chris.txt  henry.txt  scot.txt   simon.txt  
  
$ for i in *.txt ; do  
> mv -v "$i" "${i%.txt}.html"  
> done
```

The shell will return this:

```
chris.txt -> chris.html  
henry.txt -> henry.html  
scot.txt  -> scot.html  
simon.txt -> simon.html  
  
$ ls  
  
chris.html  henry.html  scot.html   simon.html
```

Of course, we haven't set the filetype properly, but we can do that pretty easily now:

```
settype -t text/html *.html
```

Type `settype -h` in a Terminal window for details on using the `settype` command.

The shell supports a bunch of different replacement and substitution commands, and they're all just as “easy” to remember as the % substitution (see Table 8.1). Some of them are pretty esoteric, and you won't end up using them very often, if ever. Still, it's handy to know they exist when you need to manipulate command-line arguments or other strings in your shell scripts. It's always faster to use the shell's built-in substitutions than to use another command like `sed`.

Table 8.1 Shell Replacement

and Substitution Commands

Command

`${parameter:-word}`

Description

If `parameter` isn't set, or is empty, return `word`; otherwise return `parameter`. This can be handy if you want to check an environment variable and provide a sensible default if it hasn't been set.

`${parameter:offset}`

Return a substring of `parameter` starting at `offset`. For example, if `TEXT` is set to "hello world", `echo ${TEXT:5}` will print "world"; the first six characters (because you start at 5 and the first character is offset 0) are skipped. If `offset` is negative, it's used as an offset from the end of `parameter`; `echo ${TEXT:-3}` should print "rld" because we're getting three characters from the end of `$TEXT`.

`${parameter:offset:length}`

Return a substring of `parameter` starting at `offset` and going for `length` characters. `echo ${TEXT:5:3}` will print "wor", which is three characters starting at offset 5.

`${#parameter}`

Return the number of characters in `parameter`.

`${parameter#word}`

If `word` matches the *beginning* of `parameter`, return `parameter` with `word` deleted from the beginning. For example, `echo ${TEXT#hello}` will return "world". This is the opposite of the `%` substitution that we used earlier.

`${parameter%word}`

If `word` matches the end of `parameter`, return `parameter` with `word` deleted from the end. We've already used this one.

`${parameter/pattern/string}`

If `pattern` matches part of `parameter`, return `parameter` with `pattern` replaced by `string`. For example, if `FOO` is set to "eeeeek", `echo ${FOO/e/E}` will return "Eeeeeek". The pattern can include shell wildcards (see Chapter 6, *The Terminal*).

`${parameter//pattern/string}`

If `pattern` matches part of `parameter`, return `parameter` with *all* instances of `pattern` replaced by `string`. Using `FOO` again, `echo ${FOO//e/E}` will return "EEEEEk".

Command Substitution

You've already learned how to make the output of one command function as the input of another (in the *Redirection* section of Chapter 6, *The Terminal*), but what if you want to use the output of one command as an *argument* to another command? That's a subtle distinction, but consider this: What if you've got a file that lists all the files you want to run through in a `for` loop? You could look in the list of files and type everything out on the command line, but that's too much work.

Why not just embed the command you'd use to display the list? You can actually embed one command inside another one:

```
#!/bin/sh
for file in $(cat list) ; do
    something
done
```

The command between `$(` and `)` is run, and its output is used as an argument, or even as a command with arguments of its own. In this case, `cat` will print out the list of files, and the `for` loop will run through them. But you could also have a command stashed in a file somewhere and run it with:

```
$(cat /tmp/some_file)
```

If `/tmp/some_file` had `ls /boot` in it, you'd see a listing of the files and directories in `/boot`.

<<Production: the ``command`` here *must* have back-ticks, not single smart quotes - chrish thanks :) -sh>>

Using Backticks Sometimes you might see ``command`` instead of `$(command)`; these are equivalent, but the first form, which uses backticks (```), is “deprecated.” That's a geek way of saying, “Don't use this.” Using the `$(command)` form will also save you from Quoting Hell, and it's much easier to tell there's a subcommand in there.

Doing Tests

Sometimes you'd like to execute part of a script depending on something else, such as whether a file exists or whether an environment variable is set. This is done by using an `if` statement to test whether the relevant condition is true. The general form for a simple `if` statement is

```
if TESTS ; then
    COMMANDS
fi
```

If the `TESTS` turn out to be true, the `COMMANDS` between the `then` and `fi` are executed. In this sort of construct “true” is defined as a number that isn't 0, a string that isn't empty, or a zero return value (a.k.a. an “exit status”) from a program or script.

0 the number and 0 returned by a program are different; the 0 number can be typed right into your script, but exit status values are a little different; the system keeps track of these. When a program sends back an exit status of zero, it means that everything worked.

<<< prod: sidebar starts here... >>>

A Bit about the Exit Status

Every command that you run in the Terminal sends back an “exit status” when it finishes to let the shell know whether it succeeded or not; this number is kept hidden by the shell (you won't see it printed in the Terminal). This idea of an exit status is a little strange at first, but you can test it yourself. Every BeOS system comes with a couple of commands named `true` and `false`; these commands don't do anything but return an exit status. The `true` command's exit status is 0, and `false`'s is something else. It doesn't actually matter what else, as long as it's not 0.

You can try these out in an `if` statement:

```
$ if true ; then
> echo we got true
> fi
we got true

$ if false; then
> echo we got false
> fi

$
```

You can use the `true` and `false` commands anywhere you'd normally use a test.

<<< prod: sidebar ends here >>>

<<prod: format as note>>

The most common command to use as one of the `TESTS` is, oddly enough, `test`, which returns an exit status of `true` if its test succeeds, or `false` if it doesn't. For example, the `test` command to check to see if a string isn't empty is `test -n`; you can use this to see if an environment variable is set or not:

<<<prod: make sure the “-n” doesn't split over two lines>>>

```
$! /bin/sh
if test -n "$FOLDER_PATH" ; then
    echo "FOLDER_PATH is set"
fi
```

If `FOLDER_PATH` is set to something, you'll see "FOLDER_PATH is set" in your Terminal.

You can also form the `test` command using square brackets. This lets you write the test for `FOLDER_PATH` like this:

```
$! /bin/sh
if [ -n "$FOLDER_PATH" ]; then
    echo "FOLDER_PATH is set"
fi
```

Most people think it's much easier to read the version with square brackets, so I'll be using them throughout the rest of this chapter.

<<end note>>

Other Commands for Tests Any command can be used as a test in the `if` statement. Properly written command-line tools will have an exit status of `true` if they succeed and `false` if there's an error.

<<<prod: format as Note>>>

Remember, these exit status values are kept hidden by the system; you won't actually see the words "true" and "false" appearing in your Terminal after running a command.

<<<prod: end Note>>>

For example, this command:

```
#!/bin/sh
if chmod +w filename ; then
    echo "Made filename writeable."
else
    echo "Had an error."
fi
```

will print "Made filename writeable." if the `chmod` command succeeds, or "Had an error." if it fails (which will happen if `filename` doesn't exist). This can be handy if you want to do something special when a command fails or display a custom error message.

An `if` statement can be more complex, too:

```
if TESTS_1; then
    COMMANDS_1
```

```

elif TESTS_2 ; then
    COMMANDS_2
...
else
    COMMANDS_N
fi

```

Each additional set of tests and commands is attached with an `elif` (short for “else if”) statement. If none of the tests succeeds, the commands in the `else` statement will be executed. The `else` statement is optional.

Testing with One Argument Using the `test` command, or its more readable cousin `[...]`, you can test quite a few things, such as whether a file exists, what type of file it is, and whether a string is empty or not (see Table 8.2).

To test one argument, like whether a file exists, you’d use

```
test op argument
```

or

```
[ op argument ]
```

where `op` is the kind of test. For example, `test -e` is used to see if a file exists, so you can check to see if there’s a file named `bozo` in `/boot` with

```
test -e /boot/bozo
```

or

```
[ -e /boot/bozo ]
```

Just running the `test` command like this isn’t very useful, so you’d stick it inside an `if` statement:

```

#!/bin/sh
if test -e /boot/bozo ; then
    echo "/boot has a bozo"
else
    echo "no bozo"
fi

```

Of course, being so smart and friendly, you’d want this to be more readable, so you’d use this version instead:

```
#!/bin/sh
```

```

if [ -e /boot/bozo ] ; then
    echo "/boot has a bozo"
else
    echo "no bozo"
fi

```

? Are those spaces before and after the brackets really necessary? Could you write the test like this instead?

?

```
<<<Prod: format as note>>>
```

?

? If you try this, you'll get back an error message like "[-e, command not found." The spaces aren't just there to make the script more readable, they're actually necessary — the shell can't understand the test without them. The shell can be *very* picky about syntax sometimes.

```

if [-e /boot/bozo]; then
<<<prod: end of note>>>

```

**Table 8.2 Tests
for One
Argument**

<i>Test</i>	<i>True If</i>	<i>Comments</i>
-d FILE	FILE exists and is a directory.	
-e FILE	FILE exists.	This will suc link.
-f FILE	FILE exists and is a normal file (i.e., not a directory or symbolic link).	
-L FILE	FILE exists and is a symbolic link.	See Chapter links.
-n STRING	STRING isn't empty (that is, has at least one character inside, even if that character is a space or a tab).	A string can variable or s checking wh
-r FILE	FILE exists and is readable by you.	
-s FILE	FILE exists and is not empty (that is, has more than 0 bytes of data inside; file attributes don't count).	
-w FILE	FILE exists and you can write to it.	
-x FILE	FILE exists and you can execute it.	
-z STRING	STRING is empty (that is, has no characters inside).	

Test Quick Reference

If you need a quick reminder to help you find the test you're looking for, try typing `help test | less` in a Terminal window. It's a good idea to pipe it into `less` because the help message for `test` is really long!

Using File Tests Let's try out a few of the tests from Table 8.2 in the Terminal to see how they really work. Open up a Terminal and `cd` to `/boot`. Now type this:

```
$ if [ -e beos ] ; then
> echo beos exists here
> fi
```

When you hit Enter after typing `fi` you should see “beos exists here,” which is obviously true if your system managed to boot. In this example the `-e` flag inside of the square brackets performs an existence test on its argument “beos” and tells us that, indeed, there is a directory here named `/boot/beos`. Let's see what else we can learn about it:

```
$ if [ -f beos ] ; then
> echo beos is a normal file
> elif [ -d beos ] ; then
> echo beos is a directory
> else
> echo beos is an unknown kind of file
> fi
```

You'll see that “beos is a directory” (well duh, we already knew that).

Testing with Two Arguments There are also some tests that work in pairs and take two arguments. These are used for comparing two files, two text strings, or two numbers (see Table 8.3).

To compare the dates of two files, for example, you'd use:

```
test argument1 op argument2
```

or the ever-popular:

```
[ argument1 op argument2 ]
```

where `op` is the kind of test. For example, `-nt` is used to see if the first argument is newer than the second. To see if `/boot/beos` is newer than

/boot/home (this would tell you if the system has been updated since being installed), you'd do this:

```
#!/bin/sh
if [ /boot/beos -nt /boot/home ] ; then
    echo "This system was probably updated."
fi
```

**Table 8.3 Tests
for Two
Arguments**

<i>Test</i>	<i>True If</i>	<i>Comments</i>
NUMBER1 -eq NUMBER2	NUMBER1 equals NUMBER2.	You can use see how in C
NUMBER1 -ge NUMBER2	NUMBER1 is greater than or equal to NUMBER2.	
NUMBER1 -gt ARG2	NUMBER1 is greater than NUMBER2.	
NUMBER1 -le ARG2	NUMBER1 is less than or equal to NUMBER2.	
NUMBER1 -lt ARG2	NUMBER1 is less than NUMBER2.	
NUMBER1 -ne NUMBER2	NUMBER1 isn't equal to NUMBER2.	This is handy a shell script specific strin
FILE1 -nt FILE2	FILE1 is newer than FILE2 according to the modification date and time.	
FILE1 -ot FILE2	FILE1 is older than FILE2 according to the modification date and time.	
STRING1 = STRING2	The strings are the same.	

Say you're writing a script and you want its error messages to behave differently depending on the `SCRIPT_ERRORS` environment variable. In the documentation, you let the user know that they can set this to "polite" or "stressed" depending whether they want to see calm or overwrought error messages.

In your script, you'd handle this by doing something like this:

```
#!/bin/sh
if [ "$SCRIPT_ERRORS" = "polite" ] ; then
    echo "Your files have been deleted. Sorry."
elif [ "$SCRIPT_ERRORS" = "stressed" ] ; then
    echo "ARGH! My life is over, I killed your files..."
else
    echo "Hey, SCRIPT_ERRORS is wrong; please set it to:"
```

```
        echo "polite or stressed."
    fi
```

Checking the Exit Status The arithmetic tests (`-eq`, `-ge`, `-gt`, `-le`, `-lt`, and `-ne`) are handy for checking the exit status of another command in a script. You'll remember from *A Bit about the Exit Status* that every command sends an exit status back to the shell when it finishes

? <<Production: don't let any of those - and the characters following them split over a line!>>

to tell the shell whether it succeeded or not.

The exit status of the last command can be found in the magic environment variable `$?`, which always contains the exit status of the last command executed by the shell. By convention, an exit status of 0 means that all is well, and anything else is an error:

```
#!/bin/sh
if [ $? -ne 0 ] ; then
    echo "oh no, an error"
else
    echo "everything is good"
fi
```

In this example we're testing the value of one argument, whatever is currently in the `$?` variable, against the value of 0 by using the `-ne` test. `-ne` asks if they're not equal, so if the current value of `$?` is not 0 then the script prints "oh no, an error." This `if` statement lets scripts respond to errors.

Protecting the `$?` Variable Because the value of the `$?` variable can change as the script runs, the best way to preserve a particular exit status is to store it in another variable. This is important because even minor steps occurring later in the same script will overwrite the value of `$?` with a new exit status. Even adding an `elif` clause to the `if` statement will overwrite the value of `$?` with the exit status of the test at the start of the `if`!. In this next example we store the value of `$?` in another variable (called `return_value`), so it doesn't get changed while our script is running.

```
#!/bin/sh

# Save the exit status of the last command in return_value.
return_value=$?

# Now we can check for specific values in the return_value;
# This assumes that the last command returns an exit status
# of 1 when it can't find a file, and anything else is a
# general error
```

```

if [ $return_value -eq 0 ] ; then
    echo "everything is good"
elif [ $return_value -eq 1 ] ; then
    echo "file not found"
else
    echo "some other error"
fi

```

Testing the Opposite

If you need to test the *opposite* of something, you can use `!` to negate a test. Maybe you'd like to know if a file isn't a directory, but don't care if it's a normal file or a symbolic link:

```

#!/bin/sh
if [ ! -d /boot/beos ] ; then
    echo "/boot/beos isn't a directory"
else
    echo "/boot/beos is a directory"
fi

```

Here we use the `-d` test to see if the argument in the test is a directory but add the `!` symbol, which makes the test return `true` if the opposite of `-d` is found. These kinds of negative tests are most often used with file test operators; The string and arithmetic tests already exist in negative versions. There's no reason for you to type

```
if [ ! ARG1 -eq ARG2 ] ;
```

when you could use

```
if [ ARG1 -ne ARG2 ] ;
```

The second version is easier to read and easier to type.

Something Useful

So far, we've gone over the `for` loop, how to do substitutions, how to do tests, and the `if` statement. We really ought to be able to make something useful now, right?

A couple of our earlier examples involved taking a bunch of “unknown” files we got off the Web and giving them an extension so that other, less fortunate operating systems might be able to guess what to do with them.

The examples also pointed out a problem: We could screw this up pretty easily if we blindly assumed all the files were the same kind.

Wouldn't it be better if we could add a file extension based on the actual MIME type of the file? If we're downloading piles of stuff from the Internet with a browser like NetPositive, all the files will have correct MIME types. Since these files are usually coming from a system that doesn't have MIME types for files, they'll probably have file extensions already, so we won't have to worry about them.

Unless the files we're interested in are part of NetPositive's cache.

Say you've been surfing the Net for a while, and `/boot/home/config/settings/NetPositive/NetCache` is full of files with not-very-helpful names like `981234...1`, `981234...2`, etc. Right now, mine's got a little over 1400 files in it, going up past `981234...2000`. Yikes! What if I wanted to keep all the HTML documents in there and give them a `.html` extension so I could take them over to another system? It's going to be a real pain to go through 1400 files in a Tracker window, selecting only the ones that have an HTML document icon. (Astute readers will note that I could probably use a command-line query to find all the HTML documents in the `NetCache` directory, but that's not the point of this example, and I'd still have to rename them all by hand.)

Before I do anything, I'll copy all of the files out of the `NetCache` directory and into another one; if I wanted to clean out the cache at the same time, I'd just move them. Then I'll go about designing a shell script that will take the following steps for every file, saving me a ton of work:

- 1) If the file doesn't exist, or it's not a normal file, skip it.
- 2) Get its filetype.
- 3) If it doesn't have a filetype, use `mimeset` to try to give it one.
- 4) If the filetype is `text/plain`, give the file a `.txt` extension.
- 5) If the filetype is `text/html`, give the file a `.html` extension.
- 6) If it's still unclear what the file is, delete it.

This would all be annoying if we were just typing commands into the shell, but it's not too bad in a shell script. In fact we can do all of this using the techniques we learned in the sections above. Here's the complete listing:

```
#!/bin/sh
#
# Give file extensions to files we care about, and delete files we
# don't, based on their MIME filetype.
```

```

#
# First we create a loop that goes through each of the arguments we
# supplied to the script. In this example, we'll be passing in the
# files from the NetPositive cache, but you could run this with any
# files you wanted.
#
# Remember, $@ is a special variable that has all of the command-line
# arguments inside.

for i in "$@" ; do

    # Check to see if the file exists; since we're working on
    # command-line arguments, the user could've typed in some files
    # that don't exist.
    #
    # The -f test checks to see if a file exists; ! -f checks to
    # see if a file doesn't exist.

    if [ ! -f "$i" ] ; then
        echo "$i is not a file, skipping"

        # The continue statement continues our for loop with the
        # next argument; we want to go on with the next file
        # instead of going on down into the rest of the script.

        continue
    fi

    # This next complicated line uses the catattr command to get
    # the file's type. catattr prints out too much information,
    # so we pipe its output through awk to strip off everything
    # we don't care about.
    #
    # The "2> /dev/null" redirects any errors to /dev/null; if
    # the file has no MIME type, catattr will print an error, but
    # we don't want to see it.
    #
    # Another thing to note is the \ at the end of the line;
    # this tells the shell that we're not done with our command
    # yet. Both of these lines get combined by the \ to do what
    # we want.

    file_type=$(catattr BEOS:TYPE "$i" 2> /dev/null | \
        awk '{ print $5; }')

    # If there's no filetype, try to assign one. You'll
    # remember that -z checks to see if a string is empty, so
    # if the variable file_type is still empty, the file has
    # no type.

    if [ -z "$file_type" ] ; then

        # The mimeset command asks the BeOS Registrar to assign
        # a MIME type to the specified file.

```

```

        mimeset -f -all "$i"

        # Now the file will have a type, so we'll do what we
        # did before to read the filetype.

        file_type=$(catattr BEOS:TYPE "$i" 2> /dev/null | \
            awk '{ print $5; }')
    fi

    # Now we check to see if the filetype is one we like:
    # By adding more elif... statements, you can extend this to
    # handle other kinds of files.

    if [ "$file_type" = "text/html" ] ; then

        # Rename our HTML documents.

        mv $i $i.html

    elif [ "$file_type" = "text/plain" ] ; then

        # Rename our text files.

        mv $i $i.txt

    elif [ "$file_type" = "application/zip" ] ; then

        # Rename our zip files.

        mv $i $i.zip

    else

        # Delete anything we didn't care about.

        rm $i
    fi
done

```

Save this script as `renamer` and make it executable with `chmod`. Now you can type

```
$ renamer 981234*
```

to automatically go through the `NetCache` files giving them reasonable file extensions based on their types. Unfortunately, this doesn't change the fact that the filenames are totally incomprehensible to anyone who isn't `NetPositive`. You win some, and you lose some....

Horrible Truths about Unix

One of the evil things about Unix shells is that the buffer (or memory area) used to pick up command arguments is a fixed size; if you try to feed too many arguments to a command, it'll either ignore the ones at the end or behave strangely. If you're trying to run hundreds of files through the `renamer` script discussed here, they're going to overflow this command buffer and only a few files will actually get fed through the script.

There's a way around this, but you'll have to run the `renamer` script from the shell; there isn't a way to do it from the Tracker. Put `renamer` into `/boot/home/config/bin` where your shell can find it, `cd` into the directory full of files (or, if you've got TermHire installed, select the directory window and hit `Alt+Windows+T` or `Option+Command+T`).

Now you'll use the `find` and `xargs` commands:

```
find . -print | xargs renamer
```

This doesn't seem to do anything with files—how could it work? Well, “`find . -print`” will find every file, directory, and symbolic link in the current directory (thanks to the `.` directory argument that we're giving to `find`) and print them out (the `-print` option), producing a big list of everything in the filesystem from here down.

We pipe this list into `xargs`, which takes the input and parcels it up into chunks of commands small enough to fit into the command buffer. It passes each chunk to the specified command, which is our `renamer` script.

`xargs`'s sole purpose in life is to help you work around the Unix command buffer's inability to grow.

After using the `find/xargs` trick on my directory of over 1400 NetCache files, I'm left with “only” a couple hundred HTML and plain text files. There sure are a lot of graphics on Web pages these days, and they've all just been deleted!

“Hey, not so fast!” you scream, “There's something in that script I don't understand!”

I knew I wouldn't be able to sneak it past you. I've introduced one new thing in this script: the `continue` statement. `continue` lets you skip over the rest of the loop and continue with the next run through. We do this right away if the file isn't a regular file, since we don't want to mess with any directories that we may encounter. If the file is something other than a regular file, the `renamer` script will print a warning message, then hit the `continue` statement and go on with the next file.

Getting the MIME Type The only other tricky thing in the script is getting the file's MIME type:


```
file_type=$(catattr BEOS:TYPE "$i" 2> /dev/null | awk '{ print $5; }')
```

This just *looks* tricky; if we split it up a little it'll make more sense. The `file_type` variable is going to be set to whatever is returned by the embedded commands between `$(` and `)`. There are two commands inside connected with a pipe:

```
catattr BEOS:TYPE "$i" 2> /dev/null  
awk '{ print $5; }'
```

The `catattr` command will print the current file's MIME type, which is stored in a file attribute named `BEOS:TYPE`. We've redirected the standard error stream (which I'll describe in a minute) to `/dev/null`, the universal bit-bucket, because we don't want to see the error message if the file has no MIME type. The pipe sends the filetype into `awk`, which prints the fifth item.

A Word about Streams

Every command-line tool works with three streams: standard input, standard output, and standard error. Traditionally, input from the user comes in through standard input, output goes to standard output, and errors are printed to standard error. Hmm, this almost makes sense....

These streams are numbered from 0 to 2:

<<< prod: this table should go in the sidebar, too >>>

#	<i>Name</i>	<i>Geek Name</i>	<i>Redirecting</i>	<i>Piping</i>
0	Standard input	stdin	< file	command
1	Standard output	stdout	> file	command
2	Standard error	stderr	2> file	2 command

If you redirect `stderr` to a file, `stdout` is still going to send the program's output to your Terminal. This ability to redirect `stdout` and `stderr` to different files is often used by programmers building an application with the `make` utility. Sending `stdout` and `stderr` to different files makes it easier to keep track of (and fix!) the application's bugs.

If you run `catattr BEOS:TYPE` on a file in a Terminal window, you'll see that it prints a line like this one:

```
filename : string : text/plain
```

Counting over, we can see that the fifthth item is the filetype we wanted.

So the `file_type` variable will be set to the file's MIME type if it has one, or nothing if it doesn't. Everything else in the script should be pretty easy to figure out if you've gotten this far.

Really Advanced Shell Use

Now that we've seen everything we've talked about (for loops, if statements, and substitutions) in use, we should look at a few shell script techniques that can make your life easier.

Script Arguments

As you've seen, shell scripts can have command-line arguments just like normal programs: options to control the behavior of the script, the names of files and directories to operate on, or both.

You've also seen that you can access these arguments from a special variable—`$@`—and loop through them using a for loop. There's a lot more you can do with a script's command-line arguments, though!

Positional Parameters A script can access the first ten arguments (starting with the script's name and ending with the ninth argument) using `$0`, `$1`,...`$9`. These are called “positional parameters” by shell freaks because they're referred to by their position in the command line. You can test this by creating a simple script like this:

```
#!/bin/sh
echo 0 is: $0
echo 1 is: $1
echo 2 is: $2
```

and running it with various fictional parameters. For example, if you named this script `EchoTest`, you could type

```
$ EchoTest hello zoomer
```

and the shell would spit back:

```
0 is: EchoTest
1 is: hello
2 is: zoomer
```

Try it a few times with different numbers of parameters. If there are fewer than three arguments, you won't get an error—you'll just get a line like “2 is:” with nothing else there.

The \$0, \$1,...\$9 variables are created for you by the shell, just like the \$@ variable. Why are we limited to ten of these? It's another artifact of the way the ancient shell handles commands, and it was probably designed this way because there are only ten digits on an English keyboard.

If you miss the `ren` command from DOS (used to rename files), you could write yourself a simple shell script to mimic it:

```
#!/bin/sh
#
# Act like the DOS "ren" command.

# If the first argument or the second argument isn't there,
# print an error message and exit.

if [ "$1" = "" ] || [ "$2" = "" ]; then
    echo "usage: ren original new"
    exit
fi

# Now rename the first argument to the second argument.
mv $1 $2
```

As you can see, I keep sneaking in new bits. There are two tests in that `if` statement, and the `||` between them means OR. The “usage” message will be displayed if the first argument is blank or if the second argument is blank.

<<< prod: sidebar starts >>>

Combining Tests

Just as you can use `||` to combine two or more tests in an OR sequence (the entire statement will be true if *any* of the tests are true), you can use `&&` for an AND sequence.

The AND sequence will only be true if *all* of the tests are true. For example, we can combine the `-e` test (does a file exist?) with the `-r` test (can I read this file?) to see if a particular file exists AND we can read it:

```
#!/bin/sh
if [ -e /path/to/the/file ] && [ -r /path/to/the/file ] ; then
    echo "The file exists and we can read it."
else
    echo "You're out of luck."
fi
```

To keep track of OR and AND, just say them: “If the file exists AND we can read it....”

Remember how programs have an exit status to indicate success or failure? You can use the `exit` statement to get out of a running shell script and return an exit status.

`exit` by itself will send back an exit status of 0, meaning everything was fine. Want to let the world know you had problems? Send back something else:

```
exit 1
```

It’s a good idea to return a different exit status for every different kind of problem your script could have. Then when you document these different exit status values, other people can incorporate your script into *their* shell scripts!

“shift” Work What if you've got more than nine parameters, though? Using `$10` won't get you the next one (the shell will use the contents of the first argument in the `$1` variable with a “0” appended to the end), but there is a way around this. The `shift` command will shift all the arguments in `$2 ... $9` down by one “slot,” and assign the next argument to `$9`. Try adding this to the end of the simple script we just created to play with arguments:

```
shift
echo 0 is: $0
echo 1 is: $1
echo 2 is: $2
```

Try running it again with different numbers of arguments. I've named mine `foo` and called it with:

```
$ foo a b c
```

and the shell prints:

```
0 is: ./foo
1 is: a
2 is: b
0 is: ./foo
1 is: b
2 is: c
```

The first parameter (“a”) has fallen off the front of the list of arguments and the other two parameters have moved down, which lets us see the last argument.

This may seem pretty pointless and complex, but it's actually useful. Imagine that you've got a shell script that takes a few options to control its behavior, and that it also takes a bunch of filenames as arguments:

```
$ myscript -v -stress ugly.txt nicer.html *.jpg
```

? <<Prod: Don't let the - in -v and -stress split over a line break.>>

For this example, we'll assume that if you use -v it must come first, then -stress, then the arguments. If you run through all of the arguments with a for loop, you'll have to check every argument inside the loop to handle the -v and -stress options:

```
#!/bin/sh

# Before we start the loop, we should set up some defaults.
# We'll use an empty variable to mean "this is off"; if the user
# turns them on, we'll set them to "yes".

verbose=
stressed=

# Run through the arguments...

for arg in "$@" ; do

    # Check for the -v option.

    if [ "$arg" = "-v" ] ; then
        verbose="yes"

        # Now we've handled this, let's go on to the next.

        continue
    fi

    # Check for the -stress option.

    if [ "$arg" = "-stress" ] ; then
        stressed="yes"

        # Now we've handled this, let's go on to the next.

        continue
    fi

    # Do your work on anything that isn't an option.

    ...
done
```

This makes the loop through the arguments more complex, and you're testing for the `-v` and `-stress` options every time, whether you've seen them already or not.

By using the `shift` statement, we can do this outside the `for` loop, and still run through the arguments using the `$@` variable:

```
#!/bin/sh

# Before we start the loop, check for the -v and -stress options.

# If the first argument is -v, we'll set the verbose variable to yes.
# Then the shift statement will kick -v out of the list of arguments,
# and move the rest of them down one "slot".

if [ "$1" = "-v" ] ; then
    verbose="yes"
    shift
fi

# Do the same for "-stress". If -v was the first argument before,
# -stress will be the first after the shift statement.

if [ "$1" = "-stress" ] ; then
    stress="yes"
    shift
fi

# Now we can loop through the rest of the arguments; the -v and -stress
# options will have been removed from the list of arguments by the
# shift statements.

for arg in "$@" ; do

    # Do your work on anything that isn't an option.

    ...

done
```

In a “real” shell script with complex arguments like this, you'd have two loops. The first one would deal with all of the options and use `shift` to remove them from the list of arguments. The second loop would then run through the remaining arguments to do the work.

The Whole Shebang We've already used `$@` to go through a script's arguments, but you can also access a script's arguments with `$*`. When `$@` and `$*` appear by themselves, they behave the same, but if you put them in quotes ("`$@`" and "`$*`"), you get different results.

When the shell sees "\$@" it runs through the parameters as if each one were a separate item; with "\$*" it treats all of the parameters as one line separated by spaces. To test this, put the following in a script and try it out:

```
#!/bin/sh
echo "First with @..."
for i in "$@" ; do
    echo $i
done

echo "Now with *..."

for i in "$*" ; do
    echo $i
done
```

The first loop will print the arguments one at a time on separate lines. The second version will print all of the arguments together on one line. This could be useful if you wanted to pass the arguments on to another script or program, but it could also cause problems if you were working with files and directories that had spaces in their names. Mixing the shell with files that have spaces in their names is tricky and best avoided if possible.

<< prod: format as POWER TIP >>

There is a safe, reliable way of dealing with files and directories that have spaces in their names, but to show it to you I've got to use two things you haven't seen before. Don't worry, we'll talk about them soon; I wanted to let you know about this since I brought up filenames with spaces.

Find a directory that has some filenames with spaces in them and type this into the Terminal:

```
$ ls -1 | while read arg ; do
> echo "The file is named: $arg"
> done
```

You'll get back a list of the files:

```
The file is named: a long filename with spaces in it
The file is named: another crazy filename
```

The `while` loop runs as long as something is "true"; in this case, as long as the `read` statement can read a line of text and assign it to the variable named `arg`. What text? Well, the list of files that the `ls -1` command is printing (that argument is a 1); the `-1` option says "give me the list of files, one per line." Of course, you could get a list from a file, or the `find` command, or another script, etc.

The `while` loop is covered below in the *More Looping* section, and we'll talk about `read` in *Listening to the User*.

<<< prod: end of Power Tip >>>

Counting Parameters Sometimes it's handy to know how many arguments you've got, and you can find out using `$#`. This would let us simplify our DOS-like `ren` command:

```
#!/bin/sh

# Make sure we've got at least two arguments; if the number of
# arguments is less than two, print the error message and exit.

if [ $# -lt 2 ]; then
    echo "usage: ren original new"
    exit
fi

mv $1 $2
```

<<format as note>>

If you're going to test `$#`, place it *before* the `shift` command in your scripts! Arguments dropped with the `shift` command are gone for good, and `$#` will go down by one (be “decremented” in geekspeak) every time you use `shift`.

<<end note>>

A Bit of 'Rithmetic Doing math from the shell can be hard (you're better off using `expr`, which was discussed in Chapter 6), but sometimes it can be handy. For “real” math, embedding a call to `expr` using `$(expr)` will be much easier, but if you're doing something very simple like adding or subtracting, the `let` command is going to be faster because it's built right into the shell.

Let's say you want to look through your command-line arguments and count them. Try putting this in a script and running it with different arguments:

```
#!/bin/sh
count=1

for i in "$@" ; do
    echo argument $count = $i
    let count=count+1
done
```

You'll get a nice numbered list of the arguments. For example, say you named this script `testing` and ran it with three arguments. You'd see this:

```
$ testing one two three
```



```
argument 1 = one
argument 2 = two
argument 3 = three
```

Now change the `let` line to:

```
count=$(expr $count + 1)
```

and run it again. Notice how much slower it is? Despite the speed difference, you should use whichever method you find easier. Speed isn't usually a big deal when you're writing a script. You want it to do something for you, and as long as it gets done, who cares if it takes a few seconds...if you wanted speed, you'd learn to program in C or C++!

The `let` statement is pretty picky about its syntax; note the total lack of white space. Also note that you don't need a `$` to use `let` on the value of the `count` variable. Being consistent might be another good reason to use `$(expr ...)` instead of `let`, although `let` is pretty safe if you don't try anything too tricky and stick to the usual math operators of `+`, `-`, `*`, and `/`.

Note that the `let` statement only works on integers; if you try to make it work with floating-point numbers, it'll round things off:

```
$ x=1.1
$ echo $x
1.1
$ let x=x+1
$ echo $x
2
```

More Looping

If you've ever taken a computer science class, you'll know that sometimes for loops aren't your best bet. For instance, what if you want to do something until a certain condition is true, without looping through a list of arguments?

The while Loop Using the `while` loop, you can do just that. For example, if I wanted to wait around for a certain file to appear in `/tmp` (because some other application was running and would eventually create this file that I need for something else), I could use `while` like this:

```
#!/bin/sh

# Loop until /tmp/important_document exists.
while [ ! -e /tmp/important_document ] ; then
    # Do nothing for 60 seconds.
```

```
        sleep 60
done
```

This will go through the loop as long as the test is true; in this case, until a file named `/tmp/important_document` exists. Each pass through the loop, it sleeps for 60 seconds, giving other processes time to run. Then it goes back, checks for the file again, and so on. Programmers look down their noses at this and call it “busy waiting” or “polling”; it’s the equivalent of someone coming into your office every 60 seconds and saying, “Have you got that important document done yet?” Because we’re sleeping inside the loop, however, this isn’t as annoying for BeOS as it would be for you.

The form of a `while` loop is

```
while TESTS ; do
    COMMANDS
done
```

The `COMMANDS` will be run over and over until the `TESTS` are no longer “true” (in the same sense as the `if` statement we talked about earlier). The `TESTS` can be anything you'd use with an `if` statement, such as a command or one of the file, string, or arithmetic tests we talked about earlier.

We could use this and the string substitution from the *Substitutions* section to take a word and print its letters one to a line. For example, for the input “hello”, we'd write:

```
h
e
l
l
o
```

This might seem like a weird thing to put in a script, but you never know when you'll find yourself in a weird situation. Something like this should do the trick:

```
#!/bin/sh

# Go through all of the arguments...

for word in "$@" ; do

    # The x variable will be our offset into the word; we set it
    # to 0 because that's how programmers spell "first" and we
    # want to start with the first letter of the word.

    x=0
```

```

# Do this loop while x is less than the number of
# characters in this word; remember, the # substitution
# returns the number of characters in the given variable.

while [ $x -lt ${#word} ] ; do

    # Now we use this substitution to print 1 character
    # from the current word, at the current offset.
    #
    # As we learned in the Substitution section,
    # ${variable:offset:length} will give you "length"
    # characters from the contents of "variable", starting
    # at "offset".

    echo ${word:$x:1}

    # Increase the offset by one to move on to the
    # next character.

    let x=x+1

done
done

```

For every word in the arguments, this script will set `x` to 0, then enter the `while` loop (because 0 will be less than the length of the current word, which we get with the `${#word}` substitution). Each letter is printed by extracting one character from the word using `x` as an offset in the `echo` command. We increase the value of `x` by one and head back around for another pass until we've printed all the characters.

Run the script with a few words together as arguments, then change the `$@` to `$*` and try the script again with the same words. Notice the difference? You won't see anything if you're only using one word, but with more than one argument you'll now see a blank line between the words. That's because `$*` treats the entire command line as one unit, and `$@` treats it as separate words. You'll get a blank line for every space on the command line.

The until Loop The `until` loop looks almost the same as a `while` loop:

```

until TESTS ; do
    COMMANDS
done

```

In fact, `while` and `until` are exact opposites of each other. With `while` you're testing something that starts out true and becomes false later, but with `until` you're testing something that starts out false and becomes true later. You'll stay in the loop executing the `COMMANDS` until the `TESTS` are true.

Again, as with `if` and `while`, the `TESTS` can be any command or the file, string, and arithmetic tests from the *Doing Tests* section.

Mnemonic Device

If you have trouble remembering which loop is which, just say the commands in plain English: "while this is true, do something" or "until this is true, do something."

We could rewrite our letter printer using `until` like this:

```
#!/bin/sh

# Go through all of the arguments...

for word in "$*" ; do

    # The x variable will be our offset into the word; we set it
    # to 0 because that's how programmers spell "first" and we
    # want to start with the first letter of the word.

    x=0

    # Do this loop until x is greater than or equal to the
    # number of characters in this word; remember, the #
    # substitution returns the number of characters in the
    # given variable.

    until [ $x -ge ${#word} ] ; do

        # Now we use this substitution to print 1 character
        # from the current word, at the current offset.
        #
        # As we learned in the Substitution section,
        # ${variable:offset:length} will give you "length"
        # characters from the contents of "variable", starting
        # at "offset".

        echo ${word:$x:1}

        # Increase the offset by one to move on to the
        # next character.

        let x=x+1
    done
done
```

Instead of printing letters while `x` is less than the length of the word, we're going to print letters until `x` is greater than or equal to the length of the word. (Greater than or equal to is the opposite of less than.)

<<prod: format as note>>

If you're wondering why you have to use a strange constructs like `-ge` for greater than or equal to rather than the usual `>=` you learned in high school math, remember that the symbols `>` and `=` have special meanings to the shell. These constructs actually make things easier, since you don't have to worry about "escaping" them.

<<end note>>

Skipping Out and Breaking Things Sometimes when you're looping through something, you'd like to skip a trip through the loop, or stop looping altogether. For example, what if we didn't like the letter "e" for some reason, and we wanted our letter printer to skip over any "e" that it found? From the section on tests, we know how to *find* the "e", but how do we skip it?

We do it by using the `continue` statement, which we've already used in a couple of examples. When the shell's in a `for`, `while`, or `until` loop and sees `continue`, it skips back to the start of the loop and carries on with the next trip through.

We can change the letter printer as follows:

```
#!/bin/sh

# Go through all of the arguments...

for word in "$*" ; do

    # The x variable will be our offset into the word; we set it
    # to 0 because that's how programmers spell "first" and we
    # want to start with the first letter of the word.

    x=0

    # Do this loop until x is greater than or equal to the
    # number of characters in this word; remember, the #
    # substitution returns the number of characters in the
    # given variable.

    until [ $x -ge ${#word} ] ; do

        # Save the current letter in a handy variable.

        letter=${word:$x:1}

        # Check to see if it's the evil "e".

        if [ "$letter" = "e" ] ; then
```

```

        # We've found the offending letter; we
        # increase x to go on to the next letter,
        # then continue with the next trip through
        # the "until" loop.

        let x=x+1
        continue
    fi

    # If we made it past the if statement, we still like
    # this letter. Print the letter, then increase x
    # to go on with the next letter.

    echo $letter
    let x=x+1
done
done

```

Now whenever the current letter is an “e” the shell heads back to the `until` loop to get the next letter and carry on. We had to repeat the `let` statement; otherwise we'd be stuck in what programmers call an “infinite loop.” If `x` stayed the same, the current letter would still be an “e” (the same one!) the next time through the loop, so we'd head back to the start, but the current letter would still be an “e” so we'd head back to the start, and so on....

Go ahead and try this script with something like “hello there”. You’ll see this:

```

h
l
l
o
t
h
r

```

The break Statement Imagine that you hated “e” so much that you didn't even want to see the rest of any word that dared include this horrible letter. You'd rather have the `until` loop stop completely then go back for another word.

This is where the `break` statement comes in; it kicks you right out of the current loop. If we change `continue` in the letter printer script to `break`, we can remove the extra `let` statement. We won't need it anymore because we'll be jumping right out of the `until` loop and continuing with the next word.

Case Study

There's a tricky but very useful statement called `case` that lets you selectively execute some commands based on a word matching a specified pattern. It's easier to show you a `case` statement at work than to try to explain it cold, so here's an example.

Byron is writing a shell script that's going to have some options and take a bunch of files as arguments. Byron's one of my cats, so the command syntax used for running his script might end up looking something like this:

```
apply [ -attitude | -catnip | -catnap | -disdain ] objects
```

The `apply` script will apply attitude, some catnip, a cat nap, or some disdain to the objects specified on the command line. If none of these options is specified, the objects are completely ignored and nothing happens; he uses this mode a lot. This setting is used for things like store-bought toys, humans who want to play with the cat, etc. You'll find a few unfamiliar constructs in this script—I'll explain those at the end.

```
#!/bin/sh
#
# Apply some cat-like behavior to the specified objects.

# First we'll check $# (the number of arguments).

if [ $# -eq 0 ] ; then

    # This script is pretty useless with no arguments, so if $#
    # is 0, we want to print the usage message and exit.

    echo "usage: apply [ action ] objects"
    echo "The optional action can be one of:"
    echo " attitude"
    echo " catnip"
    echo " catnap"
    echo " disdain"
    exit
fi

# The default action is ignoring.

action="ignoring"

# The next line specifies the object's variable without giving
# it a value. We don't have a value for it yet, so we're just
# "initializing" it here. We do this so we can tell if the user
# remembered to include some objects to work on; if not, we
# could remind them.
```

```
objects=
```

```
# To dig through the options, we're going to loop while $1 (the current
# first option) is set to something; remember, the -n test checks to
# see if a string (in this case, the contents of $1) has one or more
# characters inside.
#
# We use shift to strip off arguments we've already dealt with, so
# $1 could be nothing if we run out of arguments.
#
# Down inside the loop we'll use the break statement when we think
# we've handled all of the options.
```

```
while [ -n "$1" ] ; do
```

```
    # Use case to match the current argument with one of the valid
    # options.
```

```
    # The case statement works like an if statement with a bunch
    # of elif clauses; in this case, we check the current argument
    # (in $1) against the valid options.
```

```
    case "$1" in
        -attitude)
```

```
        # Matched the "-attitude" option, so we set the
        # action to "attitude" and then dispose of this
        # argument with shift. Now $1 will be whatever
        # came next on the command line.
```

```
        action=attitude
        shift
```

```
        # Every pattern in a case statement ends with
        # ";;".
```

```
        ;;
```

```
    -catnip)
```

```
        # Matched "-catnip".
```

```
        action=catnip
        shift
        ;;
```

```
    -catnap | -nap)
```

```
        # Matched "-catnap" OR "-nap". Why does the
        # case statement use | for OR, instead of the ||
        # that we saw earlier for combining tests in an
```



```

        # if statement? Mostly because case is not the
        # if statement, and everything has a different
        # syntax under Unix.

        action=catnap
        shift
        ;;

    -disdain)
        action=disdain
        shift
        ;;

    -*)

        # Anything else that starts with - is invalid:

        echo "$1 is not a valid option. "
        echo "You owe me a cat treat."

        # Getting an invalid option is an error, so we
        # return an exit status of 1. Any exit status
        # that isn't 0 means "Houston, we had a problem."

        exit 1
        ;;

    *)

        # If we got this far, we're done traveling
        # through the options. We'll take all of the
        # remaining arguments and store them in the
        # objects variable, since these are the objects
        # we want to work on.

        objects="$*"

        # We've handled all of the command-line arguments
        # now, so we use break to kick us out of the
        # while loop.

        break
        ;;

done

# Let the world know what's happening.

echo Byron will now apply some $action to:
echo $objects

```

This is a little longer than the rest of the scripts we've seen, but it really saves Byron a lot of time! Now with one command, he can play with things, ignore them, or spread disdain throughout our apartment, leaving much more time for important things like sleeping and eating.

With the exception of the `case` statement, this is a pretty simple script. What did you expect from a cat? On each trip through the `while` loop, we examine the current argument (in `$1`) to see if it's one of the valid options. If it's not a valid option, the script displays an error message and exits. If the argument isn't an option (for this, if it doesn't start with a `-` it's not an option) we assume that the rest of the arguments are the objects we want to work on.

The form of the `case` statement is:

```
case WORD in
    PATTERN1)
        COMMANDS1
        ;;

    PATTERN2)
        COMMANDS2
        ;;

    ...
esac
```

Just as the `if` statement ends with `fi`, the `case` statement ends with `esac`, even though it looks like a typo.

<<prod: format as note>>

You may find the similarities and differences between `for` loops and `case` statements a bit confusing at first. In `for` loops, we use a construct like:

```
for i in *
```

where `*` represents all the files in the current directory. But in `case` statements, we use constructs like:

```
case argument in
    pattern)
```

where `pattern` represents a string we want to match. Don't confuse `pattern` with a range of files, even though they can both have wildcards! We're doing something very different here. The `case` statement is like an `if...elif...else...fi` statement.

<<end note>>

This is a little more complex than the `if` statement or the loops we've looked at. The `WORD` is compared against each of the `PATTERNS` in the order in which they appear. If `WORD` matches a `PATTERN`, the `COMMANDS` inside that pattern are executed. The patterns have a “)” character at the end, and the list of `COMMANDS` ends with two “;” characters.

The patterns in a `case` statement use the same matching rules as shell wildcards (see Chapter 6, *The Terminal*). A pattern of “*)” will match anything, and is usually the last pattern specified, so that it can handle unexpected values or defaults. Remember, all patterns end with “)”.

In the case of Byron's `apply` script, there are specific patterns to match all of the valid arguments (such as `-attitude` or `-disdain`). Looking at the `-catnap` handler, you'll see that you can include two or more patterns by separating them with a “|” symbol; this means `OR` to programmers. This lets you use `-catnap` or `-nap` if it's an emergency and you want to type fewer characters.

Why does `case` use “|” to create an `OR` sequence instead of the “||” we learned about earlier? The “|” in a pattern just means “this is a list of patterns; you can match any of them.” If `case` used the “||” syntax, you might think you could also use “&&” (for `AND`) in a pattern, which isn't possible.

The pattern after the `-disdain` handler will match anything that starts with a “-” character. If we've gotten this far through the patterns without matching, and something starts with a “-” character, it's an invalid option. The very last pattern will match anything; if we've gotten down here, this isn't an argument, so we must be looking at the first object we want to work on.

The `case` statement is very popular in GNU `configure` scripts. These extremely complex scripts are used to automatically query a system to help configure software before it gets compiled into an executable. Part of `configure` will attempt to guess the type of system you're using and turn it into a string that reflects the operating system, the OS version, and the kind of hardware, such as `beos-R4-powerpc` or `beos-R4-x86`. The script then uses `case` to do some platform-specific configuration:

```
case SYSTEM in

    # Other systems

    ...

    beos-*-powerpc)

        # do some BeOS on PowerPC-specific stuff
```

```

;;

beos-*-x86)

    # do some BeOS on x86-specific stuff

;;

beos-*-*)
    echo "Unknown architecture for BeOS"
    echo "Very cool, but you might have problems..."
;;

...
esac

```

Remember the file renaming script we wrote earlier, using the file's type to give it a standard file extension? We can simplify it using the `case` statement. The original script uses an `if...elif...else` sequence to assign the extension (assuming the current file's name is in the `file` variable):

```

#!/bin/sh
...
if [ "$file_type" = "text/plain" ] ; then
    echo "$file is plain text"
    mv "$file" "$file.txt"
elif [ "$file_type" = "text/html" ] ; then
    echo "$file is HTML"
    mv "$file" "$file.html"
else
    echo "$file is an unknown file"
    rm "$file"
fi

```

Using `case`, this becomes a little easier to read (and to extend with new types!):

```

#!/bin/sh
...
case $file_type in
    text/plain)
        echo "$file is plain text"
        mv "$file" "$file.txt"
        ;;

    text/html)
        echo "$file is HTML"
        mv "$file" "$file.html"
        ;;

    text/*)

```

```

        echo "$file is an unknown text file"
        rm "$file"
        ;;

    image/*)
        echo "$file is an unknown image file"
        rm "$file"
        ;;

    *)
        echo "$file is an unknown file"
        rm "$file"
        ;;
esac

```

I've already extended this with two handlers that match any kind of text file (the `text/*` pattern) and any kind of image file (`image/*`); this will make the error messages a little more informative. You could extend it to keep all of the JPEG images by adding something like this before the `image/*` handler:

```

image/jpeg)
    echo "$i" is a JPEG image
    mv "$i" "$i.jpg"
    ;;

```

It's got to go before the `image/*` handler, or the `image/*` handler will match `image/jpeg` and delete the file for you.

A good rule of thumb is to always put more specific patterns (like `text/html` or `image/jpeg` in this example) *before* more general patterns (like `image/*` or the match-anything `*` pattern).

Listening to the User

Complex shell scripts might need some sort of input from the user. Sometimes it's easier just to ask the user a question than to support a million command-line arguments.

The `read` command reads a line from the standard input stream (which is usually the keyboard, unless you're using a pipe or redirecting from a file; see Chapter 6, *The Terminal*, for more about pipes and redirection) and assigns it to the `REPLY` variable. As usual a line is defined as whatever you type until you hit the Enter key.

The `read` Command You can use the `read` command's `-p` option to display a prompt. For example, try typing this into a Terminal window:

```
$ read -p "Do you like fish? "
```

After you've entered your answer, which is automatically assigned to the `REPLY` variable by the shell, type this to see whether you like fish:

```
echo $REPLY
```

If you want to assign the input to another variable, include its name as an argument to `read`:

```
$ read var_name
```

You can also combine this with a prompt. The following `read` command:

```
$ read -p "Do you like fish? " fishy
```

will prompt you with “Do you like fish?” and assign your answer to the variable named `fishy` instead of the `REPLY` variable.

When more than one variable name is included in a `read` command, the first word of the input is assigned to the first variable, the second word to the second variable, etc. If there's more input than variables, everything else will be assigned to the last variable. For example, if we type “hello there world” into

```
$ read greeting rest
```

`$greeting` will be given “hello” and “there world” will be assigned to `$rest`. If there are more variables than words in the input, the extra variables will be empty. Typing only “hi” into the `read` command above will set `$greeting` to “hi” and `$rest` to nothing.

You could use this to extend the file renaming script to ask the user for the type of an unrecognized file. Right now, the script arbitrarily assigns the generic type `application/octet-stream`:

```
#!/bin/sh
...
# If there's still no type, give it a generic one:

if [ -z "$file_type" ] ; then
    settype -t application/octet-stream "$file"
    file_type="application/octet-stream"
fi
```

Let's change it to use `read` and ask the user what to do. Replace that `if` statement with this one:

```
#!/bin/sh
...
if [ -z "$file_type" ] ; then
    echo "$file is an unknown kind of file."
    echo "What type of file is it?"
    echo "(Just hit return if you don't know.)"
    read -p "The filetype is: " file_type

    # If they just hit return without entering a type,
    # the file_type variable will be empty. The -z test
    # is true if a string has no characters.

    if [ -z "$file_type" ] ; then
        file_type="application/octet-stream"
    fi

    settype -t $file_type "$file"
fi
```

Now when an unknown kind of file is encountered, the user will be given some information and prompted for a type:

```
$ renamer filename
filename is an unknown kind of file.
What type of file is it?
(Just hit return if you don't know.)
The filetype is:
```

After the `read`, we check to see if they entered something. If they didn't, we assign the generic filetype. If the user entered a type, we assign that type to the file.

Making Alerts As you saw in Chapter 6, *The Terminal*, BeOS comes with a command called `alert`. You can use `alert` to pop up a dialog box for the user to click on. `alert`'s arguments are

```
alert [type] text [button1] [button2] [button3]
```

But what we didn't show you in Chapter 6 is that you can change the type of alert icon displayed in the dialog box. The various types of alerts and their corresponding icons are shown in Table 8.4.

Table 8.4 Alert types

<i>Alert Type</i>	<i>Description</i>	<i>Icon</i>
--empty	No alert.	None.
--info	An informative alert.	A blue, 3D “
--idea	An idea.	A light-bulb.
--warning	Something you want to warn the user about.	A bright yell
--stop	Something that's very important...the user should stop what they're doing and look at this.	A red exclamation

<<<prod node: 08-alerts.tiff>>>

Figure 8.1: Alert types

All of the different alert icons being used to write a famous program, “hello world.”

The text is the message you want to appear in the dialog box. You can also specify up to three buttons; if you don't specify any buttons, the alert will have one button labeled OK in it.

If the alert command has any buttons, the command's exit status will be the button number (starting with 0), and the title of the button will be printed on the standard output channel.

Here's an example script demonstrating how to use the alert command's exit status:

```
#!/bin/sh

# Show the alert, asking the user what they'd prefer
# to drink.
#
# We direct alert's output into the bit-bucket because
# we don't want to see it; for this example, we're using
# alert's exit status.

alert "What would you like?" Coffee Tea Milk > /dev/null

# Store alert's exit status in the button variable;
# the exit status of the last command (which was "alert")
# is stored in the $? variable.

button=$?

# The first button was "Coffee"; if $button equals 0, they
# chose that button.

if [ $button -eq 0 ] ; then
    echo "You like coffee."
```



```

elif [ $button -eq 1 ] ; then
    echo "You like tea."
elif [ $button -eq 2 ] ; then
    echo "You like milk."
else
    echo "You don't like anything."
fi

```

Here's the same example script, but using the alert command's output instead of its exit status:

```

#!/bin/sh

# Show the alert, asking the user what they'd prefer
# to drink.
#
# We direct alert's output into the read statement to store
# the selected button's name in the "button" variable.
#
# You could also do this:
#
# button=$(alert "What would you like?" Coffee Tea Milk)
#
# These techniques both give you exactly the same results...use
# whichever one you prefer.

alert "What would you like?" Coffee Tea Milk | read button

# Now we compare the selected button to see what the user
# picked. Note how we're using strings now; alert prints the text
# of the selected button, and we've used read to store that text
# in the "button" variable.

if [ "$button" = "Coffee" ] ; then
    echo "You like coffee."
elif [ "$button" = "Tea" ] ; then
    echo "You like tea."
elif [ "$button" = "Milk" ] ; then
    echo "You like milk."
else
    echo "You don't like anything."
fi

```

Using the button string is easier than dealing with the exit status, though it can be easy to miss the read command that's saving the button text into a variable for us.

Using Functions

As your scripts become more complex, you'll find yourself looking for ways to keep things clear and organized, and to treat sections of your

scripts like “objects” that can be invoked from other sections. That's what functions are all about.

Functions in Scripts Take a look at the lowly echo command:

```
echo 'Hello World'
```

If you place this line in your script, “Hello World” is printed to the screen when that line is encountered. But what if your script needs to do this dozens of times, from different places? And what if instead of just a one-line echo command, you wanted to invoke a whole series of commands? You could turn that block of commands into a “function,” like this:

```
#!/bin/sh

hello() {
    echo "Hello World"
    ls -l > /boot/home/dirlist.txt
    cat /boot/home/dirlist.txt
}
```

Save the block above to a script and run it—and nothing will happen. Functions don't run themselves! In this case, you have only *declared*, but not yet *invoked* the function called `hello`. In order to make a declared function run, just enter its name on a blank line below the function. If you put `hello` on a line below the function above, then run the script again, your echo command and the other commands will be processed. This will become a very important concept as you start to build complex scripts, since it lets you define scripts within scripts, surround them in the function construct, and then invoke them from elsewhere in your scripts. In essence, it gives your scripts a small degree of “object-oriented” behavior.

<<prod: format as warning>>

You must always declare your functions *before* invoking them. Try to do it the other way around, and your script will fail with error messages.

<<end warning>>

For example, you might want to structure a complex script like this:

```
#!/bin/sh

PartOne() {
    BLOCK OF COMMANDS
}

PartTwo() {
    MORE COMMANDS
    PartThree
    A FEW MORE COMMANDS
}
```

```

}

PartThree() {
    YET MORE COMMANDS
}

# Now that our functions have all been declared, we can invoke them

PartOne
PartTwo

```

Note that we didn't invoke `PartThree` from the bottom of the script, but from within `PartTwo`. When the script runs, it will run `PartOne`, then the beginning of `PartTwo`, then `PartThree` as a “subroutine” of `PartTwo`, then finish up the rest of the commands in `PartTwo`. This structure lets you branch off from one point in your script to another, and then return to where you left off to do some more work.

Functions in the Shell In addition to using functions inside your scripts, you can also store them in memory and invoke them directly from the command line. For example, if you've got a set of commands that you use all the time, and want to be able to use them as quickly as possible without having to load another file from the disk, you can store functions in your `/boot/home/.profile`. This way they'll be loaded into memory whenever you launch a Terminal session, and can be run at any time, either directly from the command line or by invoking them from other scripts.

Shell functions can process command-line arguments just like a shell script:

```

#!/bin/sh

showargs() {
    echo "There are $# arguments:"

    for arg in "$@" ; do
        echo "$arg"
    done
}

```

This new `showargs` command isn't really that useful, but it's easy to use the command-line arguments in a function for something useful. They'll be needed any time you move a script that uses command-line arguments into a shell function.

Shell functions are specific to one script, and they won't leak out into your shell sessions; as soon as the shell exits, they're gone. The functions you define in your `.profile` are always available in a Terminal because the

shell that loaded `.profile` doesn't exit until you close that Terminal window.

For example, the `renamer` script uses the same slightly nasty-looking command several times to get the MIME type for a file:

```
#!/bin/sh
...
    file_type=$(catattr BEOS:TYPE "$file" 2> /dev/null | \
        awk '{ print $5; }')
...
```

Instead of typing this in several times, we could turn it into a shell function by putting this into the script before the `for` loop starts running through the arguments:

```
#!/bin/sh
...
get_type() {

    # Note how we use $1 to get the first argument in our
    # function; the $file variable we used above only makes
    # sense to commands in the for loop. The function stands
    # alone by itself, so it has to work with its arguments
    # instead.

    catattr BEOS:TYPE "$1" 2> /dev/null | \
        awk '{ print $5; }'
}
...
```

Now we can change the line used to get the filetype to:

```
#!/bin/sh
...
    file_type=$(get_type "$file")
...
```

This makes things a lot easier to read!

Shell functions exit by running though the last command in the function or any time they hit a `return` statement:

```
#!/bin/sh

do_something() {
    if something_bad ; then

        # Return NOW with an exit status of 1 (i.e., an error).
```

```

        return 1

        # More commands in the function...

        # At the end of the function we return with the exit status
        # of whatever the last command in the function was.
    }
...

```

The `return` statement exits the shell function and sends its argument back to the shell as an exit status. If you use `return` without an argument, it sends the return value of the last command as the exit status.

The `return` statement for functions is just like the `exit` statement for shell scripts; the difference is that `return` just ends the function, and `exit` ends the entire script.

Exit Status

Remember, an exit status of 0 means success or true, and an exit status of anything else means failure or false. The exit status of the last command is available in the special variable `$?` and you should save this in another variable if you need to do complex tests on it. To save the value of one variable into another, use something like:

```

#!/bin/sh
...
ReturnHolder=$?
...

```

`$ReturnHolder` will then contain the value of whatever `$?` was, freeing up `$?` to take another value later on. Use `return` to send back an exit status from a shell function or `exit` to send back an exit status from a shell script.

Debugging Your Scripts

If you've got a bug or a typo in one of your shell scripts, it can be pretty hard to figure out what's going on. Even if you do get a readable error message, it might be telling you that the error is at a perfectly valid line in the script.

You can see each line of a shell script as it's executed by adding `set -x` at the top of a shell script. For example, say you've got our original test script with `set -x` at the top:

```

#!/bin/sh
set -x

```

```
echo hello world
```

When you run this, you'll see

```
+ echo hello world  
hello world
```

Each line of the script (after the `set -x`) gets printed prefixed by `+` and a space.

You can also add `-x` to the `/bin/sh` magic cookie at the start of the script; this is exactly the same as using the `set -x` command:

```
#!/bin/sh -x  
echo hello world
```

BeOS Application Scripting

You've probably been wondering when I was going to finally get around to the exciting BeOS-specific application scripting stuff. Well, I wanted to make sure you had a good foundation in “normal” shell scripts before we went off into BeOS application scripting, because you’ll need everything you've just learned to make the most of it!

Don't let that scare you, though. You won't *need* to do any shell programming to do application scripting, but knowing some shell will let you do much more complex things.

With BeOS application scripting, you can do all sorts of things: open and close images to make your own slideshow, perform a series of complex transformations on a sample in an audio editing program, make two programs interact with each other, or convert the first letter of every paragraph in a document into a fancy drop-capital using a random font. You’re limited only by your own creativity (oh, and your documentation!).

When doing application scripting, it’s important to understand the distinction between objects and instances. An *object* is a kind of thing; these are the nouns we'll be scripting, like “windows” or “applications.” An *instance* is a particular object; if “window” is an object, the StyledEdit window would be an instance of a “window.” This will come up often, particularly if you're talking to programmers or tech support people about something (especially in an operating system that uses lots of object-oriented programming, like BeOS).

How It Works

As we discussed at the start of this chapter, BeOS application scripting works by sending and receiving normal system messages. Every BeOS application works with application scripting because BeOS itself handles the generic scripting messages.

<<style as note>>

The application scripting instructions and examples in this chapter are fairly generic, but some specific applications support more advanced options. Check your favorite application's documentation to see if it supports more complex application-specific scripting commands.

<<end note>>

Application scripting works with *commands*, *properties*, and *specifiers*. Commands are the actions you want to perform (like getting something, setting something, or opening a document). Properties are the things the commands act on (like window titles or documents). Specifiers are used to find the specific property to work with. If you're familiar with AppleScript (or English), you might think of these as verbs, nouns, and adjectives.

<<<prod node: 08-cmd-prop-spec.tiff>>>

Figure 8.2

In the scripting command `get title of window 0 of StyledEdit`, the command is `get`, the property is `title`, and the specifier is `window 0 of StyledEdit`. Similarly, `delete window 0 of StyledEdit` would close StyledEdit's first window.

We'll look at specific scripting commands shortly, when we talk about the key command-line utility (see *Application Scripting Tools*, this chapter).

Commands There are only six standard BeOS application scripting commands; the names given here are the “official” names for these commands, and not something you’d actually send to a running application:

- **count properties:** Counts the number of instances of a property. For example, if NetPositive had several windows open, you could ask it to count them. In this case, the “property” would be “window.”
- **create property:** Creates a new instance of a property. This is the command that would open a new document.
- **delete property:** Deletes an instance of a property. Use this to close documents or windows.
- **execute property:** Executes an instance of a property. If an application let you create a macro, this would let you run the macro.

- **get property:** Gets the value of an instance of a property. You can “get” the title of a window, for example.
- **set property:** Sets the value of an instance of a property. You can also “set” the title of a window.

All application scripts will be built using these six commands, although some specific applications may add custom commands to better support whatever it is that the app does.

Remember, these are the “official” names for these commands; scripting utilities like *hey* and scripting support in programming languages like Python will call them by different names. For example, the following table showshow the official names correspond to *hey*’s commands, and what programmers see when writing C++ code:

<i>Official Command</i>	<i>hey’s Command</i>	<i>C++ Command</i>
create property	create	B_CREATE_PROPERTY
delete property	delete	B_DELETE_PROPERTY
execute property		B_EXECUTE_PROPERTY
get property	get	B_GET_PROPERTY
set property	set	B_SET_PROPERTY

Yes, *hey* doesn’t support the execute property command. That’s not a problem, though—you can do a *lot* of useful things without it!

Properties A property is a scriptable feature of an object. Properties are given unique names within that object. For example, a window has properties named `Frame` (a rectangle representing the size and position of the window), `Title` (the text in the window's title tab), and one or more `view` properties (the contents of the window).

<<<prod node: 08-frame-title-view.tiff>>>

Figure 8.3:

The NetPositive window has several properties: the `Frame`, a `Title`, and a bunch of `View` properties.

Some properties are other objects; for example a window's `view` is actually a *view object*, which has its own set of properties.

An object can have more than one instance of a property. If an application has several windows open, it will have more than one `window` property, one for each window. Asking for *the* window in an application could be ambiguous, but asking for the first window, or a window named `Funky`

would work. It's not always enough to identify something with just a property; sometimes you need a specifier to help narrow it down.

Specifiers Specifiers let you target a specific instance of a property and come in two parts:

- The name of the property, such as `window`
- Something to identify a specific instance of a property, like its name

<<format as note>>

If you were writing a BeOS program in C++ to do your scripting, you'd have a wide range of possible identifiers, but when you're doing scripting from the shell using a command-line scripting tool like `hey`, you're limited to things like names or numbers. We'll talk more about this a little later.

<<end note>>

You can “stack” several specifiers together to help locate a specific object; for example, if you want to get the frame of the second view in a NetPositive window named “Welcome to Be, Inc.” you'd type a `hey`

? <<< production: don't let this wrap >>>

command like this:

```
hey NetPositive get Frame of View 2 of Window "Welcome to Be, Inc."
```

And you would see a result like the one shown in Figure 8.4.

<<<prod node: 08-frame-view-2.tiff>>>

Figure 8.4:

The results of asking `hey` for the `Frame of View 2 of Window "Welcome to Be, Inc."`

The specifiers go from the object you're after up through all of the objects that contain it. In this case, the frame is part of the view, which is part of the window. This is pretty natural for English speakers because it's almost exactly how we would say it if we were describing it to someone else!

Scripting Suites Being able to find the scripting abilities supported by an object lets you work with almost any kind of object, even if you're never heard of it before. Every object's scripting abilities are organized into one or more scripting “suites.” A suite is defined as a standard set of supported specifiers and their properties.

For example, every button in an application is the same kind of thing (a button with some text in it) and they all behave the same way, causing

something to happen when you click on them. The arrows in a scrollbar don't look like buttons, but they act like them by scrolling your document when clicked. If buttons and scrollbar arrows could handle the same kinds of scripting commands (which they can), this common set of commands would be the "button" suite (buttons are actually part of a larger "control" suite—a set of common scripting commands that work with every GUI control in all BeOS applications).

You can find the suites (and the specific properties) supported by BeOS objects (like windows and buttons) by looking in the BeBook, the programming documentation for BeOS that gets installed automatically on every system. At the end of every object's description is a section on scripting support, listing the supported suites. Take a look at the BWindow description in the Interface Kit, for example. Unfortunately, these documents are intended for programmers, not users trying to do some scripting, so you'll have to experiment.

Objects can support more than one suite of commands; a given object will respond properly when you send it any command from any of those suites. For example, menus support the "menu" suite and the "view" suite. This lets them work with any command from both the "menu" and "view" suites.

Suites have MIME-style names like `suite/vnd.Be-control` (for user-interface controls like buttons and checkboxes) and at the time of this writing the only suites available were those defined by Be. In the future, there will be suites of scripting commands defined by third parties (and by Be) for graphics applications, for example, which will allow all graphics applications to work with similar scripting commands.

<<<prod: format as Note>>>

Scripting suite names *look* like MIME filetypes (like "`suite/vnd.Be-control`"), but they aren't. Suite names are completely internal to BeOS, and the fact that they look like MIME types is just a geeky detail.

Don't be fooled!

<<<prod: end Note>>>

Another way of finding an object's supported suites is to use `hey's getsuites` command:

```
$ hey NetPositive getsuites
```

You'll learn more about `hey` and `getsuites` later in this chapter.

Application Scripting Tools

This is all pretty abstract, and we can't start looking at real examples until we've discussed the tools that let you take advantage of BeOS's application scripting from the shell. We're going to do our scripting from the shell using the `hey` command-line utility because it lets us take advantage of everything we learned earlier in this chapter, and because most of the other scripting languages (such as Perl) can't currently take part in BeOS message passing.

Unfortunately, there aren't very many tools available yet. Application scripting is pretty new on BeOS, and a lot of developers are still trying to figure things out. Luckily, BeOS Masters Award winner Attila Mezei has given us an excellent scripting tool in the form of `hey`, which is available on his Web pages (<http://w3.datanet.hu/~amezei/>).

<<< prod: format as Note>>>

Remember, even though we're talking about the shell and `hey` here, BeOS applications can be remote-controlled by any programming language, utility, or application that can send messages. By the time you read this, that will include Python and may include Perl—if so, you won't need `hey`, but the principles we talk about here will still apply.

<<< prod: end Note>>>

Installing `hey` After you've downloaded `hey` (make sure you get the latest version—this section assumes you're using `hey` version 1.1.1 or later), and unpacked the archive, you'll be confronted with a problem. There isn't an executable in the archive!

`hey` is distributed as source code, with documentation and two project files (one each for PowerPC and x86 systems). This is good for developers, who love to have the code for everything, but not so good for users who just want to get something done. It's not very hard to build your very own copy of `hey`, though.

1. Double-click on the appropriate project file in the `hey` folder—`hey.PPC.proj` if you're using a PowerPC-based system or `hey.Intel.proj` if you're using an x86-based system.
2. The BeIDE will pop up on your screen after a few seconds and open the project.
3. Select Make from the Project menu or hit Alt+M (or Command+M if you're on a Mac keyboard). The IDE will build a fresh new `hey` executable for you.

4. Close the BeIDE window and move your new `hey` executable (with its pile of blocks icon) into `home/config/bin` on your boot disk. You're done!

Working with hey

Just as writing a useful shell script involves stringing together a bunch of smaller shell commands, writing a script to control a GUI application involves stringing together a bunch of scripting commands.

In both cases, the individual commands only move you a little closer to your goal. It's when you stick them together that they get you where you want to go.

<<<prod: format as Tip>>>

You can duplicate all of the examples in this section using a `hey`-like set of commands in Python courtesy of an add-on called `heymodule`. Be sure to check the Languages section of BeWare for Python add-ons that support application scripting!

<<<prod: end Tip>>>

Using hey `hey` supports the standard BeOS scripting commands shown in the *Commands* section above, plus a couple of other useful verbs that are listed just below.

The syntax of a `hey` command is:

```
hey application verb [ specifier [ of specifier ... ] ] [ to value ]
```

<<format as note>>

The syntax shorthand I use above is the standard way of showing a command's options; it describes how to use a command. Arguments in square brackets are optional, so one or more specifiers are allowed, and so is the “to” clause at the end. You'll need that “to” clause when you want to set a property.

<<end note>>

The application can be specified as an application's name (what you see in the Deskbar, like “Tracker” or “ShowImage”) or as an application signature like `application/x-vnd.Be-TRAK` or `application/x-vnd.Be-ShowImage`. (Most application signatures are in the form `application/x-vnd.company.programname`; if they're not, the developer receives a visit from

the MIME Police!). Details on application signatures (or "app_sigs") can be found in Chapter 5, *Files and the Tracker*.

<<< prod: format as Tip >>>

You can find the signature of an application by dropping it onto the File Types preference window. You can read all about the File Types application in Chapter 10, *Prefs and Customization*.

<<< prod: end Tip >>>

The hey Verbs The verb can be any one of those shown in Table 8.5. The last three (quit, save, and load) aren't standard scripting commands, but they *are* standard messages that all applications know how to handle. Although they're not, strictly speaking, part of application scripting, they're definitely useful.

<i>hey Verb</i>	<i>What It Does</i>	<i>Comments</i>
get	Gets an instance of a property.	The same as
set	Sets an instance of a property.	Same as set p
count	Counts the instances of a property.	Same as cou
create	Creates a new instance of a property.	Same as crea
delete	Deletes an existing instance of a property.	Same as dele
getsuites	Gets the supported suites for a property.	This isn't a s for several α suites
quit	Quits the application.	Not a standa
save	Saves a document.	Another shor
load	Loads a document.	commands. More short:

Table 8.5
The verbs hey uses to control applications

Applications can respond to a huge number of different messages, including the standard scripting messages that we're talking about here. Some of these messages tell the app to do useful things, like quit (because the user told it to or because the system is being shut down) or save the current document. These "other" messages are usually only available to programmers working in C++, but hey makes them available to you from the shell.

Sending Any Message The verb in your hey statement can also be any four characters enclosed in single quotes, such as ' _ABR '. This lets you send *any* message, which can be handy if an application supports a useful but nonstandard command. BeOS messages have a four-character ID so applications can figure out what kind of messages they are; ' _ABR ' is

the ID for “show me the About box,” and every GUI application knows how to respond to this message. If you want to send a specific message like this, you *must* include the single quotes, or hey might think it’s one of the verbs it knows about.

<<< prod: format as Tip >>>

BeOS flings around a lot of messages totally unrelated to scripting, like the “show me the About box” message. You can find out about these by looking in the BeBook and the Application Kit’s `AppDefs.h` header (located in `/boot/develop/headers/be/app`). Most of these messages require extra information to do anything useful, which means they might cause problems if you send them using `hey`. An application expecting to find the right kind of data in the message could become confused and crash. So be careful!

<<< prod: end Tip >>>

Hey Property Specifiers The property specifiers are a little more complex. Property specifiers let you talk to and send messages to a specific object. They give you a way to let `hey` know what you’re talking to. Specifiers come in several flavors:

```
name
name [index]
name [-reverse_index]
name "instance_name"
```

(In these cases, the square brackets and quotation marks are required—they’re not part of the shell command description.)

Just specifying a property name is the easiest; for example:

```
$ hey application/x-vnd.Be-TRAK get Name
```

will return “Tracker” on my system.

The next specifier (`name [index]`) lets you specify an integer to find a specific instance. Open a couple of Tracker windows and try a few indexed commands like this:

```
$ hey Tracker get Title of Window [0]
$ hey Tracker get Title of Window [1]
```

You’ll get a reply something like this:

```
BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Tracker Status"
```

for the windows that are open. You'll notice that there are two hidden Tracker windows around all the time, Tracker Status and Desktop. These are used by the Tracker to display the “Copying files” status window and the Desktop background; don't mess with them unless you don't mind crashing. Windows are given an index based on the order in which they're opened, and the indexes only count windows that are *currently* open. If you specify an index that isn't valid, you'll get an “index out of range” error message.

For example, say you've got a few Tracker windows open, and you've just seen that the last window is named “beos” (because it's open on the /boot/beos directory). You could confirm this by typing:

```
$ hey Tracker get Title of Window [3]
```

Using a negative number will count from the last item instead of the first. You could get the title of the last window using:

```
$ hey Tracker get Title of Window [-1]
```

Instance Names Another easy specifier uses the instance name (name "instance name"). This lets you refer to instances of objects by name, which is great if you already know the name. A silly example using this technique is to open StyledEdit and type this:

```
$ hey StyledEdit get Title of Window "Untitled 1"
```

You'll see that the window named “Untitled 1” has a title of...well, I'm sure you can guess. You can use this if, for example, you've used a script to open a document (StyledEdit's window title will be the filename), and you want to do something with that document. By referring to the window by name, you can be sure you're working with the document you just opened.

Values Values are the things you'll be using in a set command, and they can usually be specified as strings (put them in quotes) and numbers. Sometimes you need to specify Boolean values (“true” or “false”), specific kinds of numbers, points, rectangles, colors, or files.

You'll have to consult the documentation for the application you're trying to script to see if its set property commands require a specific kind of value or if you can just use a plain old string or number.

To use a specific kind of value, or one of the special values (such as a color), you use this syntax:

`kind(value)`

Values in all applications can be defined or organized by type. Table 8.6 lists the various types of values a given application may support. The documentation for the application you're scripting will tell you if you need a specific kind of value for a particular object and what values are allowed.

<i>Type of Value</i>	<i>Description</i>
<code>bool</code>	A Boolean value can be either <code>true</code> or <code>false</code> . For example, to set something to true, you'd use a value of <code>bool(true)</code> .
<code>BPoint</code>	The x and y coordinates of a point on the screen, in a window, etc. BPoints take two values: <code>BPoint(x,y)</code> , where x and y are numbers.
<code>BRect</code>	A rectangle specified as the coordinates of the top-left corner and the bottom-right corner. Four numbers are needed here <code>BRect(left,top,right,bottom)</code> .
<code>double</code>	A double-precision floating-point number; double-precision floats let programs do more precise calculations, though they take up more memory.
<code>file</code>	The path to a file, directory, or symbolic link. For example, to set something to <code>/boot/some/file</code> , you'd use <code>file(/boot/some/file)</code> .
<code>float</code>	A floating-point number, such as 1.5.
<code>int8</code>	An 8-bit number; the documentation that came with your application will let you know if it needs a specific kind of number (like <code>int8</code> , <code>int16</code> , or <code>int32</code>). Most of the time, you can just use the number you want.
<code>int16</code>	A 16-bit number. A 16-bit number can be twice as large as an 8-bit number.
<code>int32</code>	A 32-bit number. A 32-bit number can be twice as large as a 16-bit number.
<code>rgb_color</code>	A color specified as red, green, blue, and alpha values from 0 to 255: <code>rgb_color(red,green,blue,alpha)</code> .

Table 8.6
Types of Values

Let's try some examples of finding and using values in simple key statements. For instance, when you start StyledEdit by double-clicking on its icon in the apps folder, it puts a blank window in the top-left corner of your screen. Let's find out what the `Frame` rectangle of that window is:


```
$ hey StyledEdit get Frame of Window [0]
```

You should get back something like this:

```
BMessage(B_REPLY):  
  "result" (B_RECT_TYPE) : BRect(7.0, 26.0, 507.0, 426.0)
```

which gives you the value of the position of that window—its size and shape, as you can see in Figure 8.x.

<<<prod node: 08-stylededit-1.tiff>>>

Figure 8.5

The BRect of an empty StyledEdit window

We can use scripting and hey's set verb to change the value and move the window down and to the right:

```
$ hey StyledEdit set Frame of Window [0] to "BRect(107,76,607,476)"
```

<<<prod node: 08-stylededit-2.tiff>>>

Figure 8.6:

Changing the StyledEdit window's Frame

This example returns a message that looks like something might've gone wrong:

```
BMessage(B_REPLY):  
  "error" (B_INT32_TYPE) : 0 (0x00000000)
```

The hey command always prints out the reply message it gets after sending your message. This reply always has one extra bit of information tagging along, the “error” value. In this case, the “error” is set to the value 0 (which we get to see as a normal number and as a hexadecimal number). As you'll remember from our earlier discussion about the exit status, 0 means “no problem!,” so all is well.

If you get an “error” other than 0, and hey doesn't translate it into English for you, take a look in the Support Kit's `Errors.h` header (found in `/boot/develop/headers/be/support`), which might help. Every standard error is defined there, and the comments help explain what they mean.

You'll note that I added 100 to the top and bottom values, and 50 to the left and right values; this is to keep the StyledEdit window the same size and shape as it was originally. By changing just one of these values (or both in another proportion), you can stretch or shrink the window.

You can also change the entire size and shape of a window by altering its `Frame` in a script. This command:

```
$ hey StyledEdit set Frame of Window [0] to "BRect(307,76,607,476)"
```

<<<prod node: 08-stylededit-3.tiff>>>

Figure 8.7:
Changing the StyledEdit window's `Frame` again; this time we change its shape, too.

lets you specify each side of the `Frame` rectangle of a window.

These scripting commands will work with *any* BeOS application that shows up in the Deskbar. Experiment for yourself to verify this!

Property Names Property names in the BeBook,, but unfortunately, they can be hard to find (they're hidden in with all of the programming documentation, in the scripting support section of most objects), and not everyone can survive wading through a long description of some code they're never going to use.

Different kinds of objects support different sets of properties (and you already know these sets are called suites). You can get a list of these properties, and some information about their contents, using `hey's getsuites` option:

```
hey NetPositive getsuites of View [0] of Window [0]
```

Something horrible like this will appear in your Terminal:

<<< prod: make sure none of these lines get wrapped>>>

property	commands	types	specifiers
	Menu PCRT (extra_data: 0x1) (CSTR,data), (LONG,what)		6 2 3
	Menu PDEL (extra_data: 0x1)		6 2 3
	Menu		6 2 3
	MenuItem PEEXE PDEL (extra_data: 0x3)		6 2 3
	MenuItem PCRT (extra_data: 0x3) (CSTR,data), (LONG,what)		6 2 3
	MenuItem (extra_data: 0x2)		6 2 3
	MenuItem PCNT	LONG	1

	(extra_data: 0x4)			
Enabled	PGET PSET	BOOL	1	
	(extra_data: 0x5)			
Label	PGET PSET	CSTR	1	
	(extra_data: 0x6)			
Mark	PGET PSET	BOOL	1	
	(extra_data: 0x7)			
property	commands	types	specifiers	
Frame	PGET PSET	RECT	1	
Hidden	PGET PSET	BOOL	1	
View	PCNT	LONG	1	
View			2 3 6	
property	commands	types	specifiers	
Suites	PGET (CSTR,suites) (SCTD,messages)		1	
Messenger	PGET	MSNG	1	
InternalName	PGET	CSTR	1	
BMessage(B_REPLY):				
"suites" (B_STRING_TYPE) : "suite/vnd.Be-menu"				
"suites" (B_STRING_TYPE) : "suite/vnd.Be-view"				
"suites" (B_STRING_TYPE) : "suite/vnd.Be-handler"				
"messages" (B_PROPERTY_INFO_TYPE) : see the printout above				
"messages" (B_PROPERTY_INFO_TYPE) : see the printout above				
"messages" (B_PROPERTY_INFO_TYPE) : see the printout above				
"error" (B_INT32_TYPE) : 0 (0x00000000)				

This output is in two parts: the table at the top, and the BMessage chunk at the bottom. Within the table, the property column lists the names of the properties in View [0] of Window [0] (like Menu and Frame). The commands column shows a short version of the scripting commands that each property can understand:

<i>Short Fform</i>	<i>hey Command</i>
PCNT	count
PCRT	create
PDEL	delete
PEXE	None (This is the execute property command that hey doesn't support.)
PGET	get
PSET	set

Lastly, the specifiers column tells you how you can use that property. The important specifiers are these three:

<i>Specifier</i>	<i>Description</i>
1	A "direct" specifier; you can use this directly in the hey commands. For

example, `hey NetPositive get Label of View [0] of Window [0]` will return that window's View's label.

- 2 An “index” specifier; you can use a number to choose a specific instance of this property. For example, you could get `Menu [0]` in this view.
- 6 A “name” specifier; you can choose a specific instance of this property by using its name. We’ve already seen this when we sent commands to a specific `StyledEdit` window with `hey StyledEdit get Frame of Window "Unknown 1"`.

The rest of the output is the reply message that `hey` always prints when it's done. In this case, there's a lot more there than just the error value (remember, 0 means “everything is OK”). The various “suites” entries are what we're interested in; they indicate what scripting suites this object understands. In this case, the View knows about the `suite/vnd.Be-menu`, `suite/vnd.Be-view`, and `suite/vnd.Be-handler` suites of scripting messages. If you guessed that the BeBook's documentation about `BHandler` (in the Application Kit) and `BMenu` and `BView` (both in the Interface Kit) would have information about these suites, you'd be right. Each of these has a scripting support section, giving the names of the properties and the message you use to access those properties. The message is always one of the standard C++ scripting commands (shown earlier in the *Commands* section), such as `B_GET_PROPERTY`.

Looking back at the `getsuites` output for `NetPositive`, we see that there's a `Label` property in the first view of the first window. Just to refresh your memory a little:

```
$ hey NetPositive getsuites of View [0] of Window [0]
property  commands          types          specifiers
-----
...
          Label  PGET PSET          CSTR          1
                  (extra_data: 0x6)
...

```

Let's find out what that label says:

```
$ hey NetPositive get Label of View [0] of Window [0]
BMessage(B_REPLY):
  "message" (B_STRING_TYPE) : "this menu doesn't have a label"
  "error" (B_INT32_TYPE) : -2147475448 (0x80002008)

```

Ah-ha! `view [0]` must be a menu (since it's telling us “this menu doesn't have a label”). We might think that we could get the menu name because the `Menu` property in the `getsuites` listing has a 2 in the specifiers column, so it understands indexes, making us conclude that `Menu [0]` will be the first, and probably only, menu inside `view[0]`. Lets try it:

```
$ hey NetPositive get Menu [0] of View [0] of Window [0]
Didn't understand the specifier(s) (error 0x80002008)
```

That didn't work; hey didn't know what to do. Let's see what the menu knows about. Looking in the BeBook's Interface Kit section under BMenu's "Scripting Support" heading, we see that menus know about several properties, as shown in Table 8.7.

<i>Property</i>	<i>Description</i>
Enabled	Specifies if the menu or menu item is enabled or disabled.
Label	The text label in the menu or menu item.
Mark	Specifies if the menu or menu item has a checkmark next to it.
Menu	If the menu has another menu inside of it (hierarchical menus, for example), it will also have a Menu property. You'd refer to this submenu with something like Menu [0] of Menu [0] of View [0] of Window [0]. Look carefully if you think this is what we just tried; there's an extra Menu [0] of... in there, which means we're looking at the menu <i>inside</i> the menu. Yes, that's confusing; BeOS menu bars are menus, and the things that pop down when you click on them are also menus.
MenuItem	The items inside the menu.

Table 8.7:

Properties in the suite/vnd.Be-menu suite

So, we should be able to get the label for the menu:

```
$ hey NetPositive get Label of Menu [0] of View [0] of Window [0]
BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "File"
```

Looking in the NetPositive window, we can see that we've found the File menu. This isn't too useful, right? I mean, we can see the File menu just by looking at the window!

But menus have an Enabled property, right? What if we set that to false?

```
$ hey NetPositive set Enabled of Menu [0] of View [0] of Window [0] \
to "bool(false)"
BMessage(B_REPLY):
```

<<<prod: format as Note>>>

If you type this all on one line, you can leave off the “\” character. That’s here in this example to let the shell know that I haven’t

finished typing this long command yet, and that it shouldn't try to run the command when I press the Return key.

```
<<<prod: end of Note>>>
```

We've just disabled NetPositive's File menu! Don't worry, you can turn it back on by setting the `Enabled` property to true. `hey`'s return message seems to be totally empty; in this case, no news is good news, and setting a menu's `Enabled` property doesn't send us back a reply message.

Menus also have a `MenuItem` property; we can use this to get the items inside a menu. `MenuItems` have text labels, so let's assume they have a `Label` property just like `Menu` does:

```
$ hey NetPositive get Label of MenuItem [0] of Menu [0] of View [0] \
of Window [0]
BMessage(B_REPLY):
    "result" (B_STRING_TYPE) : "New"
```

A quick poke into the File menu will tell you that New really is the first item. Success!

These specifiers are getting pretty huge, though; let's try moving them into a shell script where we can work in the comfort of our favorite text editor.

```
#!/bin/sh
#
# List the contents of NetPositive's File menu using application
# scripting.

# We'll stick part of this huge specifier into the target_menu variable
# to save typing.
#
# This will also make it easier to aim our script at another window;
# just change the index right here.

target_menu="Menu [0] of View [0] of Window [0]"

# Similarly, we'll store the application name in the app variable,
# to make it easier to aim this script at another application.

app="NetPositive"

# Start at the first item, which is numbered 0.

index=0

# Do this loop as long as the hey command's exit status indicates
# that all is well. When we get past the last MenuItem, hey will
# return an exit status that means "an error occurred", and then
# we'll stop looping.
```

```
while hey $app get Label of MenuItem [$index] of $target_menu ; do
    # Go on to the next MenuItem.
    let index=index+1
done
```

When you run this script a whole bunch of reply messages will be printed in your Terminal:

```
BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "New"

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Open Location..."

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Open File..."

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : ""

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Save As..."

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Close"

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : ""

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Page Setup..."

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Print..."

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : ""

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "About NetPositive"

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : ""

BMessage(B_REPLY):
  "result" (B_STRING_TYPE) : "Quit"

menu item index out of range (error 0x80000003)
```

Compare this to the contents of the File menu; we've got all of the menu items (with blank ones for the separators), but what's that error message at the end? After we've gotten to the end of the menu, we go through the

while loop again, and this time, hey fails and kicks us out of the loop because we've asked NetPositive to tell us about a menu item that doesn't exist.

This is a little ugly, though, because there's way too much information coming out of the hey command. Let's move the hey command out into a shell function and use the sed command to get just the result.

<<<prod: format as Note>>>

This is one of the reasons why a “real” programming language with scripting support is high on everyone's Christmas list. Using hey from the shell to do non-trivial things can be quite a challenge; doing this from a programming language would give you back the string you asked for, without all of this extra stuff.

For example, using the Python hey module, instead of having to deal with this reply message, you'd just get back the label of the menu item as a string, ready to use for your own evil purposes.

<<<prod: end Note>>>

```
#!/bin/sh
# This script will give us a nice, clean list of
# NetPositive's scriptable menu items.

# Create a new function for this script, to collect the menu item's
# name and display it in a nicer fashion than hey's output.

show_result() {

    # Make sure the function has the right number of arguments.
    # $# returns the number of arguments, and we want to
    # have two of them; otherwise you're not calling this
    # function properly.

    if [ $# -ne 2 ] ; then

        # Sending back anything other than 0 means we
        # had a problem. In this case, not enough
        # arguments to do anything useful.

        return 1
    fi

    # Store hey's output in the "info" variable.
    #
    # Note that NetPositive is hard-coded in here; we could
    # replace it with a variable to make this easier to change.

    info=$(hey NetPositive get Label of MenuItem [$1] of $2)
```



```

# If hey's exit status (in the $? variable) isn't 0, then
# something bad happened.

if [ $? -ne 0 ] ; then

    # Print an error message, then return with an
    # exit status of 0.

    echo "Error getting label: $info"
    return 2
fi

# Now use sed to strip off everything between the start of the
# line and the :, which happens to be the last thing before the
# data we really want.
#
# Note that sed is using a regular expression to strip off
# everything before the : character (".*" matches everything)
# by replacing it with nothing.

info=$(echo $info | sed -e "s/.* : //")

# $1 below is a positional parameter -- it refers to
# the position of the original argument to the function.
#
# This prints something like: menu File: "New"

echo menu $1: $info

# Return 0 as our exit status, meaning "all is well".

return 0
}

# The next line creates a variable to be used as "shorthand" later.

target_menu="Menu [0] of View [0] of Window [0]"

# Initialize the index variable at zero. Later, we'll increment it
# by 1 for every pass through the loop, so we can get data on each of
# the menu items sequentially.

index=0

# Now call the show_result function as long as its exit status
# indicates that all is well.

while show_result $index "$target_menu" ; do

    # Move on to the next item.

    let index=index+1

```

done

When you run this script, you'll see something like this:

```
menu 0: "New"
menu 1: "Open Location..."
menu 2: "Open File..."
menu 3: ""
menu 4: "Save As..."
menu 5: "Close"
menu 6: ""
menu 7: "Page Setup..."
menu 8: "Print..."
menu 9: ""
menu 10: "About NetPositive"
menu 11: ""
menu 12: "Quit"
Error getting label: menu item index out of range (error 0x80000003)
```

This is much nicer output. Changing this to use a function makes the script a little more complex, but we want to make sure we've got the right number of arguments at the start of the `show_result` function, and then we need to check `hey`'s exit status to make sure it worked. Putting all of those commands inside the function actually makes life easier for us. After `show_result` calls `hey` to collect its output, we have to send its output through `sed` to strip off everything we're not interested in; this uses the little trick of replacing the matched string with nothing. The `while` loop at the end then calls `show_result` instead of `hey`, passing it the current index and the target menu as arguments.

Some Examples

Now that we've fooled around with `hey` a little and found out how to determine what in our applications is scriptable (by poking around with `hey`'s `getsuites` command), let's try doing something a little more useful, and actually use `hey` to automate the GUI.

Hiding the Unhidable The PoorMan Web server has an annoying shortcoming: You can't hide its window automatically.

If you're starting PoorMan in your `UserBootscript` (so it comes up automatically every time you boot into BeOS), you'll end up either moving the PoorMan status window out of the way every time, or double-clicking on its title bar to hide it. Wouldn't it be better if you could hide it automatically when it starts?

You can do this by adding a couple of lines to your UserBootscript. Try adding these just after the line that starts PoorMan:

```
#!/bin/sh
...
# Start PoorMan

/boot/beos/apps/PoorMan &

# Give PoorMan a chance to get started, then hide its window:

sleep 2
hey PoorMan set Minimize of Window [0] to "bool(true)"
```

You might be able to sleep for one second (or no time at all), but that will depend on how fast your system can launch PoorMan. Now after PoorMan starts, its window will be minimized, and you won't have to look at it.

Where Am I? In Chapter 6, you learned that you could change the shell prompt from its default (\$) to other things, including your current directory:

```
export PS1='$PWD> '
```

For some people (I'm not naming names; you know who you are!) this isn't enough, especially if they've got several Terminal windows open. You can still forget which directory you're in by ignoring the prompt.

It might help if the title of the Terminal window changed from "Terminal 1" to "Terminal: /boot/your/current/directory" but how could you make this work?

As mentioned in Chapter 6, the `cd` command is used for moving around the filesystem. If we want to update the Terminal window every time we change directories, we'll have to make up our own version of this command to change the directory and then tell the Terminal to update its window title. Let's stick these shell functions in our `/boot/home/.profile`:

```
mycd() {
    # Change to the directory specified on the command line.

    cd "$@"

    # Use hey to set the title of the window to reflect
    # the current directory. We direct all of hey's output
    # to /dev/null because we don't want to see it every time
    # we change directories.
```

```

        hey Terminal set Title of Window [0] to "Terminal: $PWD" \
        > /dev/null
    }

```

After you reload the `.profile` (by typing `/boot/home/.profile`) you'll be able to use `mycd` to move around the filesystem and update the Terminal's title to show your current directory. We send `hey`'s output to the bit bucket because we don't want to see it. Give it a try and see what happens!

Using builtin This isn't very convenient, though; you've got to remember to use a different command every time you want to move around *and* you've got to type more characters every time. Wouldn't it be easier if we could somehow name this `cd`? The shell has a command named `builtin` that tells it to use the built-in version of a command; this is exactly what we need! Change `mycd` to look like this:

```

cd() {

    # Change to the directory specified on the command line
    # using the original built-in "cd" command.

    builtin cd "$@"

    # Use hey to set the title of the window to reflect
    # the current directory. We direct all of hey's output
    # to /dev/null because we don't want to see it every time
    # we change directories.

    hey Terminal set Title of Window [0] to "Terminal: $PWD" \
    > /dev/null
}

```

In other words, the shell has a command called `cd`, and we're defining our own function called `cd`. By using `builtin`, we can have our cake and eat it too. After you reload the `.profile` again, `cd` will update the Terminal's title bar to include the current directory.

<<prod: format as power tip>>

If you're a fan of the `pushd` and `popd` commands (see your favorite `bash` manual for details), you can easily update these scripts to work with those commands as well, just by adding a couple of extra functions as variants of the `cd` function.

If you've got more than one Terminal window open, you'll notice a slight problem. The title bar of the *first* Terminal window is being updated, even if you're working in another window! To see this, hit `Alt+N` (or

Command+N) in the Terminal; this fires up a new Terminal window. It'll load the `.profile` and use the new `cd` function. Now try using `cd` to change directories; as if by magic, the title bar of the first Terminal window tells you where the second Terminal's current directory is.

Unfortunately, there's no way to fix this, other than to limit yourself to only one Terminal window. The shell has no way of finding out *which* window it's running in. This isn't such a good thing (programmers and power users *love* having lots of Terminal windows open; I've usually got three or four going with different things happening in each one), so I've asked Be to add a feature to let the shell figure out what Terminal window it's running in.

<<<prod: format as Tip>>>

In R4, the Terminal learned a new trick: It can use the same sequence of magic characters that the X Window System's `xterm` terminal uses to set its title. This little trick lets us get around the problem of not knowing what Terminal we're using.

By using this sequence of magic characters, all of your Terminals can have titles that follow you through the filesystem! Try adding this version of `cd` to your `.profile`:

```
cd() {  
  
    # If the cd command succeeds, set the Terminal title.  
  
    if builtin cd "$@" ; then  
        echo -e "\E]0;$PWD\a"  
    fi  
}
```

The text inside between the `;` and the `\` (in this case, the contents of the `PWD` variable) will be displayed in the current Terminal's title bar.

<<<prod: end Tip>>>

Name That Picture The ShowImage application that comes with BeOS will display any image on your system, as long as you've got an appropriate Translator installed. When you use ShowImage to open an image, it uses the filename as the title for the window.

After starting ShowImage, you'll get a small, blank window with a menu. Open a Terminal and find one of the images on your hard drive. Let's get ShowImage to load the BeOS logo that gets installed in `/boot/home/SampleMedia/images:`

```
hey ShowImage load "file(/boot/home/SampleMedia/images/Be Logo 1)"
```

After a second, ShowImage will pop up a window titled “Be Logo 1” with a very nice rendered 3D Be, Inc. logo inside. This isn't nearly enough information for me, though. I've got a program called `file` installed from the GeekGadgets repository. `file` is a command-line tool that uses a set of rules to give you back information about a file's type. Usually this includes a little more information than the MIME type that you can get in the Tracker or the FileTypes application. Running `file` on this image, I can see what it is:

```
$ file "/boot/home/SampleMedia/images/Be Logo 1"
Be Logo 1: TIFF image data, big-endian
```

The `file` command works with most common types of files and can sometimes tell you useful stuff (with GIF files, for example, it'll tell you how big the image is).

Let's stick this extra info into our ShowImage window's title:

```
hey ShowImage set Title of Window "Be Logo 1" to \
"${file '/boot/home/SampleMedia/images/Be Logo 1'}"
```

We can write a little script to load a bunch of images and update their titles to show the extra file information, too:

```
#!/bin/sh

# Loop through the command-line arguments.

for name in "$@" ; do

    # Tell ShowImage to load the current file. Note that we
    # assume that ShowImage is already running.

    echo Trying to load "$name"...
    hey ShowImage load "file($name)"

    # Give ShowImage a chance to load the file. You can probably
    # tune this down a bit; it'll depend on the amount of memory
    # you've got, what other applications are running, etc.

    sleep 5

    # Now tell ShowImage to change the title of this image's
    # window. We store the output from "file" in the "info"
    # variable just to keep things clean.
```

```

echo Trying to update title for "$name"...
info=$(file "$name")
hey ShowImage set Title of Window "$name" to "$info"

# Give ShowImage a chance to update the title. If you have
# a fast system, you can probably lower this to 1, or even
# remove it completely.

sleep 2
done

```

Run this with a few images and you'll have a screen full of pictures with more information than just their file names.

Email Settings BeOS comes with a useful email server and client. Unfortunately, the basic mail client, BeMail, can only cope with one user at a time. If you've got several email accounts, you'll be running the E-mail preferences application all the time, switching back and forth between your accounts to check for new mail (for information about setting up your email, see Chapter 4, *Get Online Fast*).

This is exactly the sort of thing that application scripting can make easier. We should write a script that changes the POP username, POP password, POP host, and SMTP host (if you've got multiple email servers), as well as the “Real name” and “Reply to” settings (especially if the accounts are for multiple people). Once we've done that, we should immediately check for new email.

The E-mail preferences application only has one window, so we'll be dealing with “something of Window [0]” here. After some poking around with hey, it looks like the interesting views are inside “View 0 of Window 0” (see Table 8.8).

<i>Specifier</i>	<i>Label</i>
Label of View [0] of View [0] of Window [0]	POP username
Label of View [1] of View [0] of Window [0]	POP password
Label of View [2] of View [0] of Window [0]	POP host
Label of View [3] of View [0] of Window [0]	SMTP host
Label of View [10] of View [0] of Window [0]	Real name
Label of View [11] of View [0] of Window [0]	Reply to
Label of View [16] of View [0] of Window [0]	Check Now

Table 8.8
Finding all the interesting views in the E-mail preferences app

<<<prod: format as Tip>>>

You can easily modify the script we used to list the names of the items in NetPositive's File menu to give you a list of the views in the E-mail preferences panel. Try it! You'll need to change:

- The `show_result` function's call to `hey` (you want to get the "Label of view [\$1]" and you want to talk to E-mail)
- The `target_menu` to "View [0] of Window[0]"

<<<prod: end Tip>>>

It's probably a good idea to turn on the email status window while you're doing this, to make sure everything is working properly. You can check the "Show status window" checkbox and hit the Save button, or you can type this:

```
hey E-mail set Value of View [13] of View [0] of Window [0] to 1
hey E-mail set Value of View [18] of View [0] of Window [0] to 1
```

After a second or two the email status window will appear on your Desktop. Be sure to turn on the Log checkbox if it's not already on! With the Log turned on, you'll be able to see what's happening when your system tries to download your email.

Now that we've found the views we want to script, we need to think about what our script is going to do. A script that can take one command-line argument (for the name of the email account), set everything up, and then check for new email should be pretty useful. We're going to need to:

- 1) Check for the right number of arguments (the email account)
- 2) Get settings for all of these things based on the email account
- 3) Start the E-mail preferences application if it's not already running
- 4) Set the fields in the E-mail preferences application
- 5) Simulate a click on the Check Now button

You already know how to do all of this, believe it or not! Let's look at the script (which I've called `check_mail`):

```
#!/bin/sh
#
# Script to set up the E-mail preferences application for various
# different email accounts and check for new mail.

# Check for the right number of arguments; if there isn't one argument
# we exit with an exit status of 1 to indicate that something went
# wrong.
```



```

if [ $# -ne 1 ] ; then
    echo "usage: $0 account"
    exit 1
fi

# Now use a case statement to decide what settings to use. From the
# look of things, my cats have been using email again, and they've
# edited this script to make it easier...
#
# NOTE: some.net.com is a fictional ISP; you'll have to customize
#      this case statement to include your accounts and settings!

# Attempt to match the first (and only) argument against the
# names of the accounts we know about.

case "$1" in
    poe)
        # Settings for Poe's email account.
        pop_user_name="poe"
        pop_password="claws"
        pop_host="pop.some.net.com"
        smtp_host="smtp.some.net.com"
        real_name="Poe (Lord of the Carpet)"
        reply_to="poe@cats.net.com"
        ;;

    byron)
        # Settings for Byron's email account.
        pop_user_name="byron"
        pop_password="cantaloupe"
        pop_host="pop.some.net.com"
        smtp_host="smtp.some.net.com"
        real_name="Byron (Baddest cat in the land)"
        reply_to="byron@cats.net.com"
        ;;

    meaghan)
        # Settings for Meaghan's email account.
        pop_user_name="meaghan"
        pop_password="furball"
        pop_host="pop.some.net.com"
        smtp_host="smtp.some.net.com"
        real_name="Meaghan the Cutie"
        reply_to="meghan@cats.net.com"
        ;;

    chris)
        # Settings for Chris's email account.
        pop_user_name="chrish"
        pop_password="funkburg3r"
        pop_host="pop.some.othernet.com"
        smtp_host="pop.some.othernet.com"
        real_name="Chris Herborth"
        reply_to="chrish@gnx.com"
        ;;

```

```

lynette)
    # Settings for Lynette's email account.
    pop_user_name="lynette"
    pop_password="semprini"
    pop_host="pop.some.net.com"
    smtp_host="smtp.some.net.com"
    real_name="Lynette Woodward-Herborth"
    reply_to="lynette@some.net.com"
    ;;

*)

    # We didn't match any of the accounts we know
    # about, so gripe at the user and exit with an
    # exit status of 2 to indicate the kind of error.

    echo "$1" is not a valid email account
    exit 2
    ;;

esac

# Now we should start up the E-mail preferences app if it's not already
# running.
#
# We check to see if E-mail is already running by attempting to get
# its Title. We aren't actually interested in the title, so we
# send hey's output to /dev/null.
#
# The test in the if statement is to see if hey's exit status is
# not "all's well"; this will be the case if E-mail isn't running
# yet (because hey won't be able to get its window title).

if ! hey E-mail get Title of Window 0 > /dev/null ; then
    echo Starting E-mail preferences...
    /boot/preferences/E-mail &

    # Just in case you've got a slow machine...this until
    # loop will print the message and sleep for one second
    # over and over until hey is able to get E-mail's
    # window title (which means it's running). The test
    # in the "until" will only be true when hey can get the
    # title.
    #
    # Again, we don't want the output of hey (we just want its exit
    # status) so we send it to the bit bucket.

    until hey E-mail get Title of Window 0 > /dev/null ; do
        echo Sleeping for a second while E-mail starts...
        sleep 1
    done

fi

```

```
# E-mail is now up and running, so we can set our fields based on our
# account info.
#
# All of these hey commands are redirected to /dev/null because we
# don't really care about their output. The commands still do
# something (send a scripting message to E-mail), but by redirecting
# hey's output to /dev/null, we won't be interrupted by all of the
# reply messages that every hey command prints.
#
# The \ at the end of each line just tells the shell that we're
# not done with the command yet; it ties it together with the next
# line.
# This is just to make things fit nicely on the page; in your version
# of this script, you could just type it all on one line without
# the \ character.
```

```
echo Setting up email preferences for account: "$1"
```

```
# To save typing we'll use $container as shorthand:
```

```
container="View [0] of Window [0]"
```

```
hey E-mail set Value of View [0] of $container to "$pop_user_name" \
> /dev/null
hey E-mail set Value of View [1] of $container to "$pop_password" \
> /dev/null
hey E-mail set Value of View [2] of $container to "$pop_host" \
> /dev/null
hey E-mail set Value of View [3] of $container to "$smtp_host" \
> /dev/null
hey E-mail set Value of View [10] of $container to "$real_name" \
> /dev/null
hey E-mail set Value of View [11] of $container to "$reply_to" \
> /dev/null
```

```
# Now simulate a click on the Check Now button by setting the
# button's value to 1:
```

```
echo Checking for new mail...
hey E-mail set Value of View [16] of $target to 1 > /dev/null
```

```
# That's all folks! We leave with the exit status of the last
# hey command.
```

```
exit
```

```
<<<prod: power tip>>>
```

The Python `heymodule` package found on BeWare includes a Python-based version of this script. It's much easier to read, and runs a *lot* faster!

```
<<<prod: end power tip>>>
```

After you run this, you can hit the Revert button and close the E-mail preferences window to restore your original settings.

Another approach to this problem would be to find where the E-mail preferences are stored and keep one preferences file for each account. Then you could swap the preferences, fire up the E-mail preferences app, and hit Check Now to get that account's email.

I could be a total freak, but I like the scripting solution better; it doesn't cause permanent changes to anything, and I don't really have to do much. I could even set up icons on my Desktop that called `check_mail` for one of the accounts; then I could just double-click on the icon and find out if one of my cats has any extra email lying around.

Where Now?

Now that you've learned the basics of application scripting under BeOS, the best thing for you to do is fire up a few applications and experiment. As scripting takes off, you'll be able to do more and more useful things with applications via remote control, such as asking an FTP client to automatically download all new or updated files since the last time you were online.

Until then, though, you can manipulate controls in running applications, move their windows around, hide or show windows, and do a host of other things using what you've learned in this section. Be sure to read the documentation that came with your favorite applications to see what interesting scripting commands they support, and keep an eye on BeWare's Languages section for updates to scripting languages like Python and Perl that will let you write your scripts in something other than the shell.

Don't be afraid to experiment! Until programmers document their software better, you'll *need* to experiment to find the views for the controls you want to script. Luckily, this is one of those things that will improve over time. If a program includes some interesting scripting abilities, be sure to thank the developer!

Making Your Scripts Run from the Tracker

You don't always want to run a script from a Terminal window; sometimes you just want to drag and drop a file onto a script from the Tracker and have it do something, or create a double-clickable custom icon you can store on your Desktop to launch your own scripts.

Unfortunately, this may not work the way you'd expect; you won't see any output from the script, and the script probably won't be able to find your files. Even if it does produce output, it'll probably put it somewhere strange. The shell has the "current working directory" concept (see Chapter 6, *The Terminal*), and shell scripts often depend on this to function properly. Without a Terminal to run in, the script's output (including any error messages) will disappear, and nothing will seem to happen.

Luckily, BeOS developer Pete Goodeve has stepped up to help us out with this problem with his `xicon` utility (available on BeWare).

`xicon` lets you run scripts from Tracker icons as if they were regular applications. It automatically opens a Terminal window for the script to run in so you can see the output and interact with it if necessary. You can also drag and drop other icons onto the script's icon in the Tracker, and the script will then run with the dragged items as arguments.

With the help of magic cookies (covered earlier in the *Magic Cookies* section), `xicon` can run any kind of script: shell, Perl, Python, Tcl, REXX...whichever language you prefer. As with any script, the magic cookie will control the script interpreter that `xicon` uses.

Installing xicon

Download `xicon` from BeWare (both PowerPC and x86 flavors are available), unpack the archive, open the `PROGRAM` folder in the new `xicon` folder, and move the appropriate version of `xicon` to your `/boot/home/config/bin` directory.

To make sure the Registrar (see Chapter 5, *Files and the Tracker*) knows about the script filetypes used by `xicon`, you can run the `mimeset` command on the newly installed `xicon` program:

```
mimeset -f -all /boot/home/config/bin/xicon
```

This'll make things a little smoother, and your special `xicon` scripts will have nice icons in the Tracker.

You should delete the unused binary (x86 if you're running on a PowerPC system, or vice versa) right now. If you drop it in the Trash, be sure to empty the Trash right away. A small bug in the Tracker considers BeOS binaries from the other architecture to be valid executables, and you could end up trying to run the x86 version of `xicon` on your PowerPC (or the PowerPC version on your x86). As you

can imagine, this doesn't work out very well, and you'll get an error message ("Not an executable.") from the Tracker.

Testing xicon To make sure you've installed `xicon` properly, try double-clicking one of the sample scripts that came in the archive. If you run the `test` example, you should see a Terminal window like the one in Figure 8.8 pop up. (The directory displayed by `test` will probably be different and show you the full path to the `xicon` folder.)

<<<prod node: 08-xicon-tests.tiff>>>

Figure 8.8:
xicon's test example

You can also drag and drop something onto `test` to get an `ls -l` listing for that file, its filetype, and a listing of its file attributes, as shown in Figure 8.9.

<<<prod node: 08-xicon-dropped.tiff>>>

Figure 8.9:
Dropping something onto xicon's test example

Using xicon

To actually use `xicon` with a script, all you have to do is drop the script onto the `convert to xicon script` file that comes with the `xicon` archive. This does several helpful things:

- Makes the script executable (sets the `x` bit)
- Changes the script's filetype to `text/x-script.xicon`
- Sets the script's preferred application to `xicon`
- Tells the Tracker to let the script accept any type of file using drag and drop

What xicon Really Does

To do this yourself on a script named `my_script`, you'd execute a series of commands like this in a Terminal window:

```
$ chmod +x my_script
$ settype -t text/x-script.xicon my_script
$ addattr BEOS:PREF_APP application/x-xicon my_script
$ rmattr BEOS:FILE_TYPES my_script
```

(If the script doesn't have any supported filetypes, you'll get an error for that last command. That's OK—we're just making sure the script can accept any kind of file.)

Now when you drop a file on the script, a Terminal window will pop up, the script will run, then the Terminal window will vanish.

Enhancing xicon Scripts

You don't always want the Terminal to close as soon as the script is done; sometimes it's nice to see what happened, and windows that pop up and vanish quickly tend to make people think their system is about to crash. Not to mention the fact that windows flashing in and out of existence are annoying. So how can you keep the window from closing when it's running from `xicon`? The flipside of this question is, how can you tell when you're running from `xicon`?

When your shell script is launched by the Tracker using `xicon`, the `FOLDER_PATH` environment variable is set to the directory where the script lives. If your `FOLDER_PATH` isn't set to anything, you're running from a normal command line, but if it is, you're running in one of these temporary Terminal windows that `xicon` opens for you.

The window will stick around if your script is asking for input from the user (such as the `test` script, above, which asks you to “Type a return to continue”). A simple way to make sure this happens is to add these lines to the end of your script:

```
#!/bin/sh
...
if [ -n "$FOLDER_PATH" ] ; then
    read
fi
```

The `if` statement is checking to see if the `FOLDER_PATH` environment variable is set to anything. If it is, we use the `read` statement to wait until the user hits the Enter key.

To see how this works, take your `test_script` from earlier in the chapter:

```
#!/bin/sh
echo "Hello world"
```

and drop it on the `convert` to `xicon` script icon in the `xicon` folder. The `test` script's icon should change to a document with a big red plus sign on it and what looks like a grey Terminal window inside.

<<<prod node: 08-xicon-icon.tiff>>>

Figure 8.10:

xicon's special script icon

Now if you double-click `test_script` from the Tracker it'll run in a window, thanks to xicon. If you didn't see anything, your system is too fast (I'll bet you never thought you'd hear that!) and the window is vanishing immediately.

Change the `test_script` to include the `if` statement:

```
#!/bin/sh

# Print our message of peace.

echo "hello world"

# Now, if FOLDER_PATH is set to something, wait for the user
# to press Enter.

if [ -n "$FOLDER_PATH" ] ; then
    read
fi
```

<<prod: format as note>>

If your favorite editor didn't preserve the file's type (maybe it changed it to a plain text file, and the icon looks like a plain old document now), drop it on the `convert` to xicon script icon again.

<<end note>>

When you double-click the new `test_script`, you'll get a Terminal window with your friendly message in it. The window will stick around until you close it or you hit the Enter key.

<<<prod node: 08-xicon-test-script.tiff>>>

Figure 8.11:

`test_script` running in its own Terminal, courtesy of xicon

Your scripts can use this `FOLDER_PATH` environment variable to help find other files or to save some output in the directory containing the script. Use something like

```
#!/bin/sh
if [ -n "$FOLDER_PATH" ] ; then
    cd "$FOLDER_PATH"
fi
...
```


at the start of your script to change into the script's directory before you do anything else.

Learning More

If you're interested in learning more about writing shell scripts, be sure to check out some of the books mentioned in the *Learning More* section of Chapter 6, *The Terminal* (especially *Learning the Bash Shell* and *Unix in a Nutshell*).

There's also a wealth of shell scripts available on the Internet, although not many are aimed at the beginner. TrackerBase, by Scot Hacker (you may have heard of him), is full of great examples for scripting newbies; you can find TrackerBase on BeWare or at Scot's BeOS software archive at <http://www.birdhouse.org/beos/software/>.

Lurking around Usenet groups like [comp.unix.shell](#) and [comp.sys.be.help](#) isn't a bad idea either.

To get more bang for your buck with BeOS GUI application scripting, keep an eye out for scripting languages that support BeOS messaging (in BeWare's Languages section), read the documentation that comes with your applications, and encourage developers to support scripting in their applications. Again, [comp.sys.be.help](#) will be a good place to ask questions and share scripting experiences, and so will the beusertalk mailing list (<http://www.be.com/aboutbe/maillinglists.html>).

You might also find some useful tidbits (including `heymodule`, which brings BeOS application scripting to Python) on my Web site, <http://www.beoscentral.com/home/chrish/Be/>. There's a lot more there than just `heymodule`; look for a large BeOS community page listing BeOS developers, useful information, links, and lots of software.